

**PCT**WORLD INTELLECTUAL PROPERTY ORGANIZATION  
International Bureau

## INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

<b>(51) International Patent Classification <sup>6</sup> :</b> <b>G06F 12/02, 9/318</b>	<b>A1</b>	<b>(11) International Publication Number:</b> <b>WO 98/48353</b> <b>(43) International Publication Date:</b> 29 October 1998 (29.10.98)
<b>(21) International Application Number:</b> PCT/US98/07624 <b>(22) International Filing Date:</b> 21 April 1998 (21.04.98)  <b>(30) Priority Data:</b> 08/841,508                      23 April 1997 (23.04.97)                      US  <b>(71) Applicant:</b> SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, Palo Alto, CA 94043 (US).  <b>(72) Inventors:</b> O'CONNOR, James, Michael; 345 Ruth Avenue, Mountain View, CA 94043 (US). TREMBLAY, Marc; Apartment 3, 801 Waverly Street, Palo Alto, CA 94301 (US). VISHIN, Sanjay; Apartment 89, 1055 Manet Avenue, Sunnyvale, CA 94087 (US).  <b>(74) Agent:</b> O'BRIEN, David, W.; Skjerven, Morrill, MacPherson, Franklin & Friel LLP, Suite 700, 25 Metro Drive, San Jose, CA 95110 (US).		<b>(81) Designated States:</b> JP, KR, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).  <b>Published</b> <i>With international search report.</i>
<b>(54) Title:</b> WRITE BARRIER SYSTEM AND METHOD INCLUDING POINTER-SPECIFIC INSTRUCTION VARIANT REPLACEMENT MECHANISM  <b>(57) Abstract</b>  A pointer-specific instruction variant replacement mechanism facilitates an exact write barrier, i.e., a write barrier specific to pointer stores and transparent to non-pointer stores. Pointer store specific instruction replacement allows some implementations to provide an exact barrier specific to the particular set of intergenerational pointer stores that are of interest to a particular garbage collection method or combination of methods. The exact identification of pointer stores herein does not require tags encoded in-line with collected memory storage and does not require non-standard word sizes to support such tags. In one embodiment, a non-quick to quick translator cache provides pointer specific store instruction replacement. In another, self-modifying code provides pointer specific store instruction replacement. An exemplary write barrier provided in accordance with the pointer-specific instruction variant replacement mechanism of this invention affords a garbage collector implementer with support for a wide variety of garbage collection methods, including remembered set-based methods, card-marking type methods, write barrier based copying collector methods, mark-sweep methods, etc., as well as combinations thereof and combinations including train algorithm type methods to managing mature portions of a generationally collected memory space. Such a write barrier can eliminate non-pointer stores from the set of stores that are evaluated against, for example, an intergenerational pointer store trap matrix or a garbage collection page mask to determine whether or not to trap. Such a write barrier can also eliminate entries associated with non-pointer stores from remembered set or card table stores for collection time scanning of modified portions of a collected generational space.		

**FOR THE PURPOSES OF INFORMATION ONLY**

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece	ML	Mali	TR	Turkey
BG	Bulgaria	HU	Hungary	MN	Mongolia	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MR	Mauritania	UA	Ukraine
BR	Brazil	IL	Israel	MW	Malawi	UG	Uganda
BY	Belarus	IS	Iceland	MX	Mexico	US	United States of America
CA	Canada	IT	Italy	NE	Niger	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NL	Netherlands	VN	Viet Nam
CG	Congo	KE	Kenya	NO	Norway	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NZ	New Zealand	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	PL	Poland		
CM	Cameroon	KR	Republic of Korea	PT	Portugal		
CN	China	KZ	Kazakstan	RO	Romania		
CU	Cuba	LC	Saint Lucia	RU	Russian Federation		
CZ	Czech Republic	LI	Liechtenstein	SD	Sudan		
DE	Germany	LK	Sri Lanka	SE	Sweden		
DK	Denmark	LR	Liberia	SG	Singapore		
EE	Estonia						

## WRITE BARRIER SYSTEM AND METHOD INCLUDING POINTER-SPECIFIC INSTRUCTION VARIANT REPLACEMENT MECHANISM

5

### BACKGROUND OF THE INVENTION

#### Field of the Invention

The present invention relates to garbage collection, and in particular, to systems and methods for isolating generations in garbage collectors.

#### 10 Description of the Related Art

Traditionally, most programming languages have placed responsibility for dynamic allocation and deallocation of memory on the programmer. For example, in the C programming language, memory is allocated from the heap by the `malloc` procedure (or its variants). Given a pointer variable, `p`, execution of machine instructions corresponding to the statement `p=malloc(sizeof(SomeStruct))` causes  
15 pointer variable `p` to point to newly allocated storage for a memory object of size necessary for representing a `SomeStruct` data structure. After use, the memory object identified by pointer variable `p` can be deallocated, or freed, by calling `free(p)`. Pascal and C++ languages provide analogous facilities for explicit allocation and deallocation of memory.

Unfortunately, dynamically allocated storage may become unreachable if no reference, or pointer, to the storage remains in the set of root reference locations for a given computation. Memory objects that are no  
20 longer reachable, but have not been freed, are called *garbage*. Similarly, storage associated with a memory object can be deallocated while still referenced. In this case, a *dangling reference* has been created. In general, dynamic memory can be hard to manage correctly. In most programming languages, heap allocation is required for data structures that survive the procedure that created them. If these data structures are passed  
25 to further procedures or functions, it may be difficult or impossible for the programmer or compiler to determine the point at which it is safe to deallocate them.

Because of this difficulty, garbage collection, i.e., automatic reclamation of heap-allocated storage after its last use by a program, can be an attractive alternative model of dynamic memory management. Garbage collection is particularly attractive for functional languages, such as the JAVA™ language (JAVA is  
30 a trademark of Sun Microsystems, Inc.), Prolog, Lisp, Smalltalk, Scheme, Eiffel, Dylan, ML, Haskell, Miranda, Oberon, etc., which exhibit data sharing, delayed execution, and generally, less predictable execution orders than the procedural languages. See generally, Jones & Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, pp. 1-41, Wiley (1996) for a discussion of garbage collection and the classical algorithms therefor.

- 2 -

Three classical garbage collection methods are reference counting, mark-sweep, and copying storage reclamation. The first, reference counting, is based on maintaining a count of the number of references, e.g., pointers, to each memory object from active memory objects or root reference locations. When a new memory object is allocated and a pointer thereto is assigned, the memory object's reference count is set to one.

5 Then, each time a pointer is set to refer to the memory object, the memory object's reference count is incremented. When a reference to the memory object is deleted or overwritten, the reference count is decremented. Memory objects with a reference count of zero are unreachable and can be collected as garbage. A reference counting garbage collector implementation typically includes an additional field, the *reference count*, in each memory object and includes incrementing and decrementing support as part of new object,

10 delete object and update pointer functions.

In contrast, tracing collector methods involve traversal of reference chains through memory to identify live, i.e., referenceable, memory objects. One such tracing collector method is the mark-sweep method in which reference chains through memory are traversed to identify and mark live memory objects. Unmarked memory objects are garbage and are collected and returned to the free pool during a separate sweep

15 phase. A mark-sweep garbage collector implementation typically includes an additional field, e.g., a *mark bit*, in each memory object. Mark-compact collectors add compaction to the traditional mark-sweep approach. Compaction relocates live objects to achieve beneficial reductions in fragmentation. Reference count methods may also employ compaction.

Another tracing method, copying collection, divides memory (or a portion thereof) into two semi-spaces, one containing current data and the other containing old data. Copying garbage collection begins by reversing the roles of the two semi-spaces. The copying collector then traverses the live objects in the old semi-space, *FromSpace*, copying reachable objects into the new semi-space, *ToSpace*. After all the live objects in *FromSpace* have been traversed and copied, a replica of the data structures exists in *ToSpace*. In essence, a copying collector scavenges live objects from amongst the garbage. A beneficial side effect of

20 copying collection is that live objects are compacted into *ToSpace*, thereby reducing fragmentation.

25

Generational approaches build on the observations that (1) memory objects typically die young and that (2) tracing methods spend considerable resources traversing, copying, or relocating comparatively long-lived objects. Generational garbage collection schemes divide the heap into two or more generations, segregating objects by age, and concentrate collection efforts (or at least more vigorous collection efforts) on

30 the younger generation(s). Since the youngest generation can be small, garbage collection related pause times can, on average, be kept short. Garbage collection within a generation can be by copying, mark-sweep, or other garbage collection method. To implement a generational collector, it is vital that a mutator process, the garbage collector or some combination of both identify intergenerational pointers so they can be treated as part of a root set by the garbage collector. A mutator is a process which changes the graph of reference chains

35 through memory in the process of performing useful work, apart from garbage collection, in a computer system.

Intergenerational pointers typically arise either through mutator process pointer stores or through promotion of objects containing pointers. Promoted intergenerational pointers can easily be detected by a collector process upon promotion. However, short of scanning older generations for pointers into younger generations—a costly process—pointer stores must be trapped and recorded to detect intergenerational pointer stores. Barriers are well known and have been implemented in hardware, in software, or with operating system (typically paging system) support. See generally, Jones & Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, pp. 165-74, Wiley (1996) (discussing intergenerational pointers, write barriers, entry tables, remembered sets, sequential store buffers, page marking with hardware support, page marking with virtual memory support, and card marking).

If software techniques, such as in-line code for pointer store checking, are used, the execution time and in-line code space overheads can be significant. One example of a software write barrier is that proposed by Ungar (see David M. Ungar, *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*, ACM SIGPLAN Notices, 19(5), pp. 157-67 (1984)), which intercepted stores to check whether (1) a pointer was being stored and (2) whether the pointer was to a younger generation object and was being stored into an older generation object. If so, the address of the older generation object was added to a *remembered set*. Software barriers can impose a large amount of overhead on the operations to which they apply. For example, a software store barrier provided by in-lined code adds additional instruction latencies, e.g., to check whether a pointer is being store and whether the pointer is intergenerational, and increases the total volume of code. Such code increases may adversely affect cache performance.

An alternative to such a software barrier is to use an operating system's virtual memory page protection mechanisms to trap accesses to protected pages or to use page modification dirty bits as a map of pages potentially containing an object with an updated intergenerational pointer field. Such techniques typically defer identifications of pointer stores, and more particularly intergenerational pointer stores, from amongst all stores until collection time. However, virtual memory page sizes are not generally well suited to garbage collection service. For example, pages tend to be large as compared with objects and virtual memory dirty bits record any modification to the associated page, not simply pointer stores. As a result the costs of scanning a page for intergenerational pointers can be high.

Another alternative to an inlined code software write barrier is hardware barrier support. Although, many write barrier implementations do not discriminate between pointer and non-pointer stores, and instead simply record all writes while deferring checks for intergenerational pointers to collection time, the extensive use of hardware support for garbage collection in the Symbolics 3600 allowed efficient implementation of a page marking scheme. Three features of the Symbolics 3600 made this feasible. First, a hardware write barrier ignored any word that was not a pointer to generational data. Whenever a reference to generational memory was stored into a page, the write-barrier hardware set a corresponding bit in the garbage collection page table. Second, a tagged architecture removed the need to consider object boundaries while performing collection time checks for intergenerational pointers since pointer words could always be distinguished from

- 4 -

non-pointer words using tags. The Symbolics 3600 accommodated a 2-bit major data type tag, a 4-bit minor tag and a 28-bit address in a 36-bit word. Finally, pages were smaller—at 256 words—than typical virtual memory pages, so a page could be scanned rapidly at collection time. See Jones & Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, pp. 169-70, Wiley (1996) (discussing page marking with hardware support on the Symbolics 3600); see also Moon, *Architecture of the Symbolics 3600*, In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 76-83 (1985) (discussing stored representations of objects).

The process of identifying intergenerational pointers can require significant collection-time scanning. One improvement is to segment collected memory space (e.g., the heap) into small regions called cards. Card marking offers several advantages provided that the cards are of the right size. Since they are smaller than virtual memory pages, the amount of collection-time scanning can be reduced. On the other hand, the amount of space occupied by a card table is less than that required for a word-by-word marking scheme. In general, a bit is set unconditionally in a card table whenever a word in the card is modified. Card marking collectors must scan dirty cards for intergenerational pointers at collection time. The cost of scanning cards is proportional to the number and size of cards marked, rather than to the number of stores performed, since duplicates never arise. See Wilson and Moher, *Design of the Opportunistic Garbage Collector*, ACM SIGPLAN Notices, 24(10), pp. 23-35 (1989).

Although generational approaches can be very effective at reducing total garbage collection time and the majority of collections can be non-disruptive, collections of older generations can be disruptive. To collect these older generations of objects in a non-disruptive manner, Hudson and Moss proposed an algorithm that processes bounded-size areas of a mature object space at each collections. The algorithm is incremental in nature and guarantees eventual collection of any and all garbage. Hudson and Moss use a train analogy to describe their solution to the problem, with carriages representing bounded size areas and trains representing groups of carriages holding linked structures. The system is efficient in that it does not rely on special hardware or virtual memory mechanisms. See Hudson and Moss, *Incremental Collection of Mature Objects*, Proceedings of International Workshop on Memory Management, St. Malo France (16-18 September, 1992).

## SUMMARY OF THE INVENTION

The present invention provides systems, methods, apparatus, and computer program products embodying such systems, methods, and apparatus, for facilitating implementations of garbage collectors. In particular, the present invention provides a pointer-specific instruction variant replacement mechanism that facilitates an exact write barrier, i.e., a write barrier specific to pointer stores and transparent to non-pointer stores. Such a write barrier eliminates non-pointer stores from the set of stores that are evaluated against, for example, an intergenerational pointer store trap matrix or a garbage collection page mask to determine whether or not to trap. Such a write barrier also eliminates entries associated with non-pointer stores from

- 5 -

remembered set or card table stores for collection time scanning of modified portions of a collected generational space.

5 An exemplary write barrier provided in accordance with the pointer-specific instruction variant replacement mechanism of this invention affords a garbage collector implementer with support for a wide variety of garbage collection methods, including remembered set-based methods, card-marking type methods, write barrier based copying collector methods, mark-sweep methods, etc., as well as combinations thereof and combinations including train algorithm type methods to managing mature portions of a generationally collected memory space.

10 Pointer store specific instruction replacement allows some implementations in accordance with the present invention to provide an exact barrier specific to the particular set of intergenerational pointer stores that are of interest to a particular garbage collection method or combination of methods, including garbage collection methods hereafter developed. The exact identification of pointer stores herein does not require tags encoded in-line with collected memory storage and does not require non-standard word sizes to support such tags. In one embodiment, a non-quick to quick translator cache provides pointer specific store instruction replacement. In another, self modifying code provides pointer specific store instruction replacement.

15 In one embodiment in accordance with the present invention, an apparatus includes a virtual machine instruction processor, an instruction replacement component of the virtual machine instruction processor, and a write barrier. Instructions executable by the virtual machine instruction processor include program occurrences of a store instruction. The instruction replacement component detects the store instruction and selectively replaces a particular program occurrence of the store instruction with a pointer-specific store instruction if a store target field of the particular program occurrence resolves to a pointer-type field. The write barrier is provided by execution of the pointer-specific store instruction on the virtual machine instruction processor. In a further embodiment, the instruction replacement component includes a translator cache coupled into an instruction path of the virtual machine instruction processor. Resolution of the store target field is triggered by the translator cache in response to a program occurrence identifier no match indication. The translator cache caches a pointer-specific variant of the store instruction and associates the program occurrence identifier therewith if the resolution indicates that the store target field is of type reference. In yet another further embodiment, the instruction replacement component replaces the particular program occurrence of the store instruction by modifying an in-memory image of the particular program occurrence of the store instruction.

20 In various embodiments, the virtual machine instruction processor alternatively includes a hardware processor to directly execute at least a subset of the instructions or a software program executable on a hardware processor wherein the store instruction and the pointer-specific store instruction are executable by the software program.

- 6 -

In another embodiment in accordance with the present invention, a method for filtering pointer stores includes detecting a program occurrence of a store instruction and selectively replacing the program occurrence of the store instruction with a pointer-specific store instruction based on resolution of store target field type information for the program occurrence of the store instruction. Execution of the pointer-specific store instruction includes selective trapping in accordance with contents of a garbage collection configuration store.

In a further embodiment, the method includes executing the pointer-specific store instruction and selectively trapping the executing in accordance with first contents of the garbage collection configuration store. The garbage collection configuration store programmably encodes a write barrier to selected intergenerational pointer stores. In another further embodiment, the method includes executing the pointer-specific store instruction and selectively trapping the executing in accordance with second contents of the garbage collection configuration store. The garbage collection configuration store programmably encodes a write barrier to garbage collection page boundary crossing pointer stores.

In yet another further embodiment, the selectively replacing includes performing a lookup in an instruction translator cache using a unique identifier for the program occurrence of the store instruction. If the unique identifier matches an entry of the instruction translator cache, substituting the pointer-specific store instruction associated therewith. In still yet another further embodiment, the selectively replacing includes modifying an in-memory image of the particular program occurrence of the store instruction.

### **BRIEF DESCRIPTION OF THE DRAWINGS**

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

Figure 1 is a block diagram of an exemplary embodiment of a virtual machine hardware processor that includes support for garbage collection generation isolation in accordance with this invention.

Figure 2 depicts "builds upon" relationships between software and hardware components of a JAVA application environment including hardware processor (Figure 1) and software components of an exemplary JAVA virtual machine implementation.

Figure 3 illustrates several possible add-ons to the hardware processor of Figure 1.

Figure 4 depicts operation of a write barrier provided in accordance with an embodiment of this invention for trapping intergenerational and card boundary crossing pointer stores made by a mutator process executing on the hardware processor of Figure 1.

Figure 5 depicts a object reference (*objectref*) format in accordance with an embodiment of this invention.



- 7 -

Figure 6A depicts an object format in accordance with an embodiment of this invention.

Figure 6B depicts an alternative handled object format in accordance with an embodiment of this invention.

Figure 7 depicts one embodiment of a bytecode replacement cache employed in accordance with this invention to dynamically replace pointer-non-specific store bytecodes with pointer-specific bytecodes to facilitate trapping of intergenerational and card boundary crossing pointer stores.

Figure 8 depicts an illustrative remembered set based generational collector approach that can be supported by architectural support for garbage collection in accordance with this invention.

The use of the same reference symbols in different drawings indicates similar or identical items.

## 10 DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

The following sets forth a detailed description of the best contemplated mode for carrying out the invention. The description is intended to be illustrative of the invention and should not be taken to be limiting.

Architectural support described herein for isolation of garbage collection generations includes an intergenerational pointer store trap matrix, object reference generation tagging, a write barrier responsive the  
15 intergenerational pointer store trap matrix and object reference generation tagging, a garbage collection trap handler, and facilities for selective dynamic replacement of pointer-non-specific instructions with pointer-specific instructions with write barrier support.

In general, embodiments in accordance with the present invention may employ various aspects of such architectural support for isolating generations in a garbage collected system. Although such architectural  
20 support may be provided in hardware, in software, or in a combination of hardware and software, embodiments in which the architectural support is provided substantially in hardware will typically provide both increased performance and reduced memory requirement advantages. For this reason, an exemplary hardware virtual machine instruction processor embodiment is described herein. However, based on this description, those of skill in the art will appreciate alternative embodiments including embodiments based on  
25 software (e.g., interpreter, just-in-time compiler, etc.) implementations of a virtual machine instruction processor which fall within the scope of the claims which follow.

### A JAVA Virtual Machine Instruction Processor Embodiment

Figure 1 depicts an exemplary hardware embodiment of a virtual machine instruction processor 100, hereinafter hardware processor 100, that includes support for bounded pause time relocating garbage  
30 collection in accordance with the present invention, and that directly executes processor architecture independent JAVA virtual machine instructions. The performance of hardware processor 100 in executing

- 8 -

virtual machine instructions is typically better than high-end CPUs, such as the Intel PENTIUM microprocessor or the Sun Microsystems ULTRASPARC processor, (ULTRASPARC is a trademark of Sun Microsystems of Mountain View, CA., and PENTIUM is a trademark of Intel Corp. of Sunnyvale, CA.) interpreting the same virtual machine instructions with a software JAVA interpreter. In addition, the  
5 performance of hardware processor 100 is better than some high-end CPU's with a JAVA just-in-time (JIT) compiler. Hardware processor 100 is low cost and exhibits low power consumption. As a result, hardware processor 100 is well suited for portable applications.

Because hardware processor 100 provides a JAVA virtual machine instruction processor implementation substantially in hardware, 25-50 Kilobytes (Kbytes) of memory storage, e.g., read-only  
10 memory or random access memory, otherwise required by a software interpreter can be eliminated or alternatively allocated. Hardware support for garbage collection provides further advantages for a limited memory JAVA virtual machine implementation by reducing in-line code for garbage collection (e.g., compiler supplied read and/or write barrier support), by facilitating improved utilization of limited memory, and by reducing garbage collection overheads and pause times. In environments where the expense of a large  
15 memory is prohibitive, including, for example, an Internet chip for network appliances, a cellular telephone processor, other telecommunications integrated circuits, or other low-power, low-cost applications such as embedded processors, and portable devices, hardware processor 100 is advantageous.

Even in environments where large memory is viable, hardware support for garbage collection reduces overheads associated with barrier implementations, facilitates improved utilization of memory, and reduces  
20 pause times for relocating garbage collector implementations. In particular, hardware processor 100 provides advantages for garbage collection methods and implementations in the context of an exemplary JAVA virtual machine implementation. However, based on the description herein, those of skill in the art will recognize variations for other JAVA virtual machine implementations, including e.g., interpreted and JIT compiler JAVA virtual machine implementations, as well as for other non-JAVA virtual machine implementations.

As used herein, a virtual machine is an abstract computing machine that, like a real computing machine, has an instruction set and uses various memory areas. A virtual machine specification defines a set of processor architecture independent virtual machine instructions that are executed by a virtual machine implementation. In general, a virtual machine implementation may be in hardware (e.g., as in the case of hardware processor 100), in software (e.g., as in the case of interpreted and JIT compiler implementations), or  
30 in hardware and software. Each virtual machine instruction defines a specific operation that is to be performed. The virtual machine need not understand the computer language that is used to generate virtual machine instructions or the underlying implementation of the virtual machine. Only a particular format for virtual machine instructions needs to be understood. In an exemplary embodiment, the virtual machine instructions are JAVA virtual machine instructions. Each JAVA virtual machine instruction includes one or  
35 more bytes that encode instruction identifying information, operands, and any other required information.

- 9 -

In this embodiment, hardware processor 100 (Fig. 1) processes the JAVA virtual machine instructions, which include bytecodes. Hardware processor 100 directly executes most of the bytecodes. However, execution of some of the bytecodes is implemented via microcode. Lindholm & Yellen, *The JAVA™ Virtual Machine Specification* (Addison-Wesley, 1996), ISBN 0-201-63452-X, which is incorporated  
5 herein by reference in its entirety, includes an exemplary set of JAVA virtual machine instructions. The particular set of virtual machine instructions supported by a hardware processor 100 is not an essential aspect of this invention. However, in view of the virtual machine instructions, those of skill in the art can modify the invention for a particular set of virtual machine instructions, or for changes to the JAVA virtual machine specification.

10 In one embodiment, hardware processor 100 includes an I/O bus and memory interface unit 110, an instruction cache unit 120 including instruction cache 125, an instruction decode unit 130 including non-quick to quick translator cache 131, a unified execution unit 140, a stack management unit 150 including stack cache 155, a data cache unit 160 including data cache 165, and program counter and trap control logic 170. Support for garbage collection features described herein resides primarily in integer unit 142 and registers 144 of  
15 execution unit 140 with some additional support in program counter and trap control logic 170 (including e.g., support for forcing the program counter to a next JAVA virtual machine instruction following a trapping store). In one embodiment, non-quick to quick translator cache 131 facilitates pointer-specificity for hardware write barrier logic of integer unit 142. Each of these units is described below.

Figure 2 depicts a "builds upon" relationship between software and hardware components of a JAVA  
20 application environment such as, for example, an application environment partially defined by and partially executable on hardware processor 100 (Fig. 1). JAVA application/applet software 210 exploits software components defining an applet/application programming interface 220 including AWT classes 241, net and I/O classes 242, and JAVA OS windows 243, JAVA OS graphics 248, TCP 244, NFS 245, UDP 246, IP 247, Ethernet 222, keyboard 249, and mouse 221 software components, which in one embodiment include JAVA  
25 bytecodes. In the embodiment of Figure 2, JAVA OS graphics 248 and Ethernet 222 software components also include extended bytecodes beyond those defined by the baseline JAVA Virtual Machine Specification. Components of an embedded application programming interface (EAPI) 230 include foundation classes 231 and hardware and software components of JAVA virtual machine implementation 250 in accordance with the JAVA Virtual Machine Specification.

30 JAVA virtual machine implementation 250 includes hardware processor 100 and trap code executable thereon to evaluate JAVA virtual machine instructions. In addition, JAVA virtual machine implementation 250 includes hardware support for extended bytecodes (including e.g., pointer store bytecodes and memory access barriers described below in the context of garbage collection); class loader 252, byte code verifier 253, thread manager 254, and garbage collector 251 software, and microkernel 255. JAVA virtual  
35 machine implementation 250 includes a JAVA virtual machine specification compliant portion 250a as well as

- 10 -

implementation dependent portions. Although the JAVA virtual machine specification specifies that garbage collection be provided, the particular garbage collection method employed is implementation-dependent.

Architectural features for garbage collection described herein in the context of an exemplary hardware processor 100 embodiment of JAVA virtual machine implementation 250 are particularly adapted for generational garbage collection methods. However, based on this description, those of skill in the art will recognize the application of bounded-pause time support of this invention to relocating collectors in general, including e.g., non-generational collector implementations, incremental mark-compact collectors, copying collectors, etc.

Figure 3A illustrates several possible add-ons to hardware processor 100 to create more complicated system. Circuits supporting any of the eight functions shown, i.e., NTSC encoder 301, MPEG 302, Ethernet controller 303, VIS 304, ISDN 305, I/O controller 306, ATM assembly/reassembly 307, and radio link 308 can be integrated into the same chip as hardware processor 100 of this invention.

In addition, those of skill in the art will appreciate a wide variety of computer systems incorporating hardware processor 100, including embodiments of hardware processor 100 with any of the above-described add-on circuits. An exemplary computer system 300 embodiment includes physical memory storage (e.g., RAM and/or ROM), computer readable media access devices (e.g., disk, CD-ROM, tape, and/or memory technology based computer readable media access devices, etc.), input/output device interfaces (e.g., interfaces for keyboard and/or pointing devices, for display devices, etc.), and communications devices and/or interfaces. Suitable communications devices and/or interfaces include those for network- or telephony-based communications, for interfacing with communications networks including land-line and/or wireless portions of a public switched network, private networks, etc. In some embodiments of this invention, instruction streams (including e.g., JAVA bytecodes) are transmitted and/or received for execution by hardware processor 100 via such communications devices or interfaces.

### **Architectural Support for Garbage Collection**

Hardware processor 100 provides architectural support for a variety of garbage collection methods, including generational collector methods implemented as garbage collection software executable thereon. In particular, hardware processor 100 includes programmable store filtering, tagged object reference and object header formats, and extended bytecode support.

#### **Programmable Store Filtering**

Figure 4 depicts one embodiment of a supervisor-writable register GC\_CONFIG that supports programmable filtering of stores to the heap. In the context of Figure 1, register GC\_CONFIG, is included in registers 144 and is accessible to execution unit 140. In one embodiment, 12 bits of register GC\_CONFIG define a field GC\_PAGE\_MASK for use in selecting a page size for inter-page pointer store checks. The 12

- 11 -

bits of field **GC\_PAGE\_MASK** are used as bits 23:12 of a 32-bit garbage collection page mask, with an additional 8 more-significant bits defined as 0x3F and 12 less-significant bits defined as 0x000. The resulting 32-bit garbage collection page mask is used to create a store barrier to pointer stores that cross a programmable garbage collection page boundary. Both the store data *value* and the *objectref* target of a pointer store (e.g., an *aputfield\_quick* instruction operating on *value* and *objectref* residing at the top of an operand stack represented at stack cache 155) are effectively masked by the 32-bit garbage collection page mask and compared to determine if *value* (itself an *objectref*) points to a different garbage collection page than that in which the target object resides. In this way, the garbage collection page size is independent of virtual memory page size. Furthermore, garbage collection pages can be provided in computer system and operating system environments, such as in low-cost, low power portable device applications or internet appliance applications, without virtual memory support. In the embodiment of Figure 4, register **GC\_CONFIG** allows programmable definition of a garbage collection page size ranging from 4 KBytes to 8 Mbytes, although, based on this description, suitable modifications for other garbage collection page sizes and size ranges will be apparent to those of skill in the art.

Register **GC\_CONFIG** also includes a field **WB\_VECTOR** for programmably defining an intergenerational pointer store trap matrix. Field **WB\_VECTOR** encodes generation pairs for which a pointer store having a pointer data generation associated with the store data *value* and a target object generation associated with the *objectref* target thereof will trap. In the embodiment of Figure 4, field **WB\_VECTOR** effectively defines a 4x4 matrix of 1-bit indications for pointer data and target object generation pairs for which a garbage collection trap is desired. Other embodiments may define larger or smaller matrices and may optionally encode additional states (e.g., *no\_trap*, *gc\_notify1*, *gc\_notify2*, and *gc\_notify3*).

Referring to the embodiment of Figure 4, two bit tags of the target object's *objectref* are concatenated with the two tag bits of the *value* being stored thereinto to form a 4-bit index into field **WB\_VECTOR**. Each of the two bit tags encodes generation membership information as described below. The indexed bit of field **WB\_VECTOR** then encodes whether a write barrier traps the corresponding pointer store. Field **WB\_VECTOR** can be programmably configured to encode trapping of all intergenerational pointer stores (i.e., of all pointer stores where the pointer data generation and the target object generation are not equal), of all pointer stores (intergenerational as well as within a generation), of younger generation pointer stores (i.e., stores of a younger generation pointer into an older generation object), etc. In general, arbitrarily complex trapping of generation pairs is supported. In the embodiment of Figure 4, up to 4 generations are supported, although based on this description, those of skill in the art will appreciate suitable modifications for larger numbers of generations.

In one embodiment, field **GC\_PAGE\_MASK** and field **WB\_VECTOR** programmably define the behavior of a write barrier to pointer stores. In particular, as described in greater detail below, extended bytecode support provided by hardware processor 100 allows such a write barrier to identify pointer stores from amongst non-pointer stores, and fields **GC\_PAGE\_MASK** and **WB\_VECTOR** further allow hardware

- 12 -

processor 100 to programmably filter the write barrier to trap a programmably defined set of pointer stores. However, alternative embodiments may forgo the advantageous exact identification of pointer stores provided by the extended bytecode support of hardware processor 100 while still exploiting programmably defined sets of apparent pointer stores in a conservative barrier implementation.

5           Figure 4 depicts programmable store filtering support in the context of a four generation collected memory space 450. Garbage collector process 420 includes bytecodes executable on hardware processor 100 for implementing a generational collector in which remembered sets 460 record younger generation pointer stores made by mutator process 410. Exemplary contents of intergenerational pointer store trap matrix 470, which correspond to the contents of field WB\_VECTOR, encode a write barrier to a younger generation  
10   pointer stores. Tags, which are encoded as described below, for the generation associated with a store data pointer value and the generation associated with the *objectref* target of a pointer store instruction (e.g., of a *aputfield\_quick* instruction) of mutator process 410 are used to index into intergenerational pointer store trap matrix 470. Based on the exemplary contents of intergenerational pointer store trap matrix 470 element so indexed, write barrier 430 traps the *aputfield\_quick* pointer store if tags associated with  
15   value and *objectref* indicate that a reference to a younger generation object is being stored into an older generation object, invoking garbage collection trap handler (*gc\_notify*) 440.

Based on the description herein, those of skill in the art will appreciate a variety of suitable implementations for garbage collection trap handler 440 which support the particular programmably selected store filtering provided by contents of intergenerational pointer store trap matrix 470 (i.e., of field  
20   WB\_VECTOR) and/or field GC\_PAGE\_MASK. In one embodiment in accordance with the exemplary contents (Fig. 4) of intergenerational pointer store trap matrix 470, garbage collection trap handler 440 includes bytecodes executable on hardware processor 100 to store information about the trapping store to a remembered set data structure (including e.g., remembered sets 461, 462, and 463 respectively corresponding to generations 3, 2, and 1 of collected memory space 450).

25           In another embodiment, contents of intergenerational pointer store trap matrix 470 are programmably defined to trap all pointer stores (intergenerational or otherwise). A related card marking type alternative embodiment of garbage collection trap handler 440 includes bytecodes executable on hardware processor 100 to store information about the trapping store to a card table data structure. In contrast with traditional card marking implementations, the architecture support for garbage collection described herein allows hardware  
30   processor 100 to distinguish pointer stores from stores in general, and to distinguish intergenerational pointer stores from pointer stores in general. For these reasons, in another card-marking type embodiment, contents of intergenerational pointer store trap matrix 470 are programmably defined to trap only intergenerational pointer stores.

35           In both the remembered set style embodiment and the card marking style embodiment, extended bytecode support allows hardware processor 100 to exactly identify pointer stores from amongst non-pointer

- 13 -

stores as described in greater detail below. Additionally, the programmable filtering of stores to the heap provided by intergenerational pointer store trap matrix 470 (i.e., by field **WB\_VECTOR**) and write barrier 430 allows hardware processor 100 to identify pointer stores which are intergenerational *at mutator process* 410 *store time*, rather than at collection time. For this reason, the term card marking is descriptive of the kind of storage (e.g., a card table) provided by an embodiment of garbage collection trap handler 440 for use by collector process 420 during collection. Usage of the term "card marking" does not imply that all stores need be trapped regardless of whether pointer or literal data is stored.

Figure 4 also depicts support for garbage collection page based trapping of pointer stores. Field **GC\_PAGE\_MASK** provides the programmable mask for comparing a store data pointer *value* and the *objectref* target of a pointer store instruction (e.g., of a `aputfield_quick` instruction). Write barrier 430 traps the pointer store if the garbage collection pages for *value* and *objectref* do not match. The additional store filtering providing by field **GC\_PAGE\_MASK** and write barrier 430 is particularly useful for collector process 420 implementations on hardware processor 100 in which Hudson's train algorithm (see generally, R. Hudson and J.E.B. Moss, *Incremental Garbage Collection for Mature Objects*, Proceedings of International Workshop on Memory Management, St. Malo, France, 16-18, Sept. 1992) is employed for non-disruptive collection of an oldest generation by an embodiment of collector process 420. Those of skill in the art will recognize suitable implementations wherein garbage collection page size defined by field **GC\_PAGE\_MASK** is used to define train "cars" in accordance with Hudson's train algorithm.

By way of example, the syntax of the `aputfield_quick` bytecode and operation of hardware processor 100 in accordance therewith is as follows:

Operation: Set a reference field in object with garbage collection checks

Format:

<code>aputfield_quick</code>
<code>offsetbyte1</code>
<code>offsetbyte2</code>

Stack: ..., *objectref*, *value*  
...

Description: The *objectref*, which must be of type `reference`, and *value*, which must also be of type `reference`, are popped from the operand stack. The *value* is written to the specified offset into the class instance referenced by *objectref*. The value of the offset is  $(\text{offsetbyte1} \ll 8) \mid \text{offsetbyte2}$ .

Runtime Exceptions: If the *objectref* is null, `aputfield_quick` throws a `NullPointerException`.

The most significant two bits of each of the *objectref* and *value* are concatenated to form a four-bit index. This index selects a bit from the 16-bit **WB\_VECTOR** field of the **GC\_CONFIG** register. If the selected bit is set, a `gc_notify` trap is generated.

- 14 -

```

index=((objectref & 0xC0000000)>>28) | (value>>30)
if(GC_CONFIG.WB_VECTOR[index]==1)
    generate gc_notify

```

Notes: The opcode of this instruction was originally putfield, operating on a field determined dynamically to have an offset into the class instance data corresponding to a field of type reference.

When the constant pool entry referenced by a putfield instruction is resolved, the offset for the field it references is generated, and the type of the field is determined. The 16-bit offset replaces the two operand bytes of the original putfield instruction. The type of the field determines whether a putfield\_quick, putfield2\_quick, or aputfield\_quick bytecode replaces the original putfield bytecode.

The syntax of an aputstatic\_quick bytecode for setting a static reference field in a class (rather than a reference field in an object) and the operation of hardware processor 100 in accordance therewith are analogous.

### 15 Fast Garbage Collection Trap Handler

In one embodiment, fast handling of garbage collection is provide by a vectored trap, gc\_notify (tt=0x27), priority=17

implemented by trap control logic 170 which triggers an associated garbage collection trap handler 440 including bytecodes executable by hardware processor 100. In one embodiment, a single garbage collection trap handler 440 services each of the above garbage collection traps, whether generated in response to a garbage collection page boundary crossing pointer store or intergenerational pointer store. Since a garbage collection trap occurs before the trapping store is completed, garbage collection trap handler 440 needs to emulate the trapping store in addition to garbage collection functions performed, such as remembered set or card table updating, in order to prevent hardware processor 100 from infinitely trapping. Garbage collection trap handler 440 should then force the PC to the instruction following the store.

JAVA virtual machine instructions affected by garbage collection traps include aputfield\_quick, aputstatic\_quick, astore, anewarray, multianewarray, newarray, putfield, putstatic, and new. Of these, only aputfield\_quick, aputstatic\_quick, and astore need to perform garbage collection checks, such as dynamic filtering in accordance with contents of fields WB\_VECTOR and/or GC\_PAGE\_MASK described above. The others need only be aware of the garbage collection mechanisms used, for example, properly initializing generational membership tags. In one embodiment, the astore bytecode traps to an emulation routine and the astore trap handler performs the appropriate garbage collection checks. One straightforward trap



- 15 -

handler implementation gets the *arrayref* being stored onto the top of the operand stack in the *aastore* trap handler and executes the bytecode sequence: *dup, getfield\_quick #0, aputfield\_quick #0* to perform the appropriate checks and trap if necessary.

### Tagged Reference and Object Formats

5           Figure 5 depicts one embodiment of an object reference (*objectref*) as represented in hardware processor 100. Three bits of the *objectref* can be used for garbage collection hints. In particular, a field **GC\_TAG** forms part of an index into register field **GC\_CONFIG.WB\_VECTOR** to determine whether write barrier 430 traps a pointer store as described above. In the embodiment of Figure 5, field **GC\_TAG** encodes generation membership information for use by write barrier 430 as described above. An additional handle bit  
10   H indicates whether the object is referenced by the *objectref* directly or indirectly-through a handle. Handles provide a referencing method that facilitates, albeit at the cost of an additional level of indirection, relocation of memory objects without large-scale updates of pointers (or *objectrefs*) thereto. Both of these fields are masked out before being provided to integer unit 142 (Fig. 1) of hardware processor 100.

          In one embodiment of hardware processor 100, an object 600 is represented in memory including a  
15   header portion 610 and an instance variable storage portion 620. Figure 6A depicts one such embodiment. Header portion 610 includes a 32-bit word that itself includes a method vector table base portion 612 for representing object's class and five bits of additional storage 614 reserved for synchronization status of the object and information for the garbage collector. Optionally, a second header-word, e.g., monitor pointer 616,  
20   can contain the address of a monitor allocated for the object, thereby making all five bits of additional storage 614 in the first header word available for garbage collection information. In the embodiment of Figure 6A, an object reference (*objectref*) points to the location of method vector table base portion 612 to minimize the overhead of method invocation.

          Three bits of header portion 610 are available to a garbage collector such as collector process 420. In header portion 610, three lower-order-bits (header[2:0]), and two high-order-bits (header[31:30]) are masked  
25   off when the header is treated as a pointer. Three of these bits (header[31:30, 2]) are available to the garbage collector to store information about object 600. Bits 1 and 0 may be used to hold LOCK and WANT bits for object synchronization. Alternatively, a second header word, e.g., monitor pointer 616, can be provided for maintaining the synchronization status of object 600, leaving all five bits for garbage collection support. How the bits for garbage collection support are used depends on the particular type(s) of garbage collection  
30   methods implemented collector process 420 and garbage collection trap handler 440. Possible uses include mark bits, counter bits to age objects within a generation, etc. As described above, in an optional second header-word embodiment of header portion 610, five bits are available to a garbage collector such as collector process 420.

- 16 -

In the embodiment of Figure 6A, instance variable storage portion 620 begins one word after the method vector table base portion 612 and contains instance variables of object 600. The least significant bit of an *objectref* specifies whether the reference is a handled ( $\text{==1}$ ) or not ( $\text{==0}$ ). An alternative, "handled," object format is depicted in Figure 6B. A handled reference is established when object 600b is created and all subsequent references go through the handle, i.e., storage pointer 650b to access the object. This support is provided for some types of garbage collector which reduce costs of object relocation by copying handles rather than the underlying object storage, including that for instance variables.

Extended Bytecode Support for Dynamic Replacement of Pointer Non-Specific Bytecodes

Hardware processor 100 includes features for accelerating execution of JAVA bytecodes by dynamically replacing bytecodes supplied to an execution unit with quick variants thereof. However, as now described, quick bytecode replacement features can also be employed by hardware processor 100 to dynamically replace data-type non-specific store operation bytecodes with pointer-specific store operation bytecodes so as to facilitate implementation of a write barrier for garbage collection.

In one embodiment, *putfield* and *putstatic* bytecodes for setting a field in an object or class are dynamically replaced with corresponding quick variants (e.g., *putfield\_quick*, *putfield2\_quick*, or *aputfield\_quick*, and *putstatic\_quick*, *putstatic2\_quick*, or *aputstatic\_quick*). The particular replacement bytecode depends on the type of field being operated upon. For example, a *putfield* bytecode which is determined to operate on a field of type *reference*, is dynamically replaced with the *aputfield\_quick* bytecode. Quick bytecode replacement takes advantage of loading and linking work done the first time the associated non-quick bytecode is executed as described in the above-incorporated reference, but more importantly for garbage collection, dynamic replacement with pointer-specific quick bytecodes allows a virtual machine instruction processor such as hardware processor 100 to differentiate between pointer storing and non-pointer storing program occurrences of a data type non-specific store bytecode. Replacement with pointer-specific bytecodes reduces the frequency of trapping stores because only pointer store bytecode variants (i.e., *aputstatic\_quick* or *aputfield\_quick*) need participate in a write barrier implementation.

One embodiment of dynamic bytecode replacement is now described with reference to Figure 7. Figure 7 is a block diagram of a portion of a hardware processor 100 which includes an operand stack 723 which in one embodiment is represented in stack cache 155 (see Fig. 1), instruction decoder 135, non-quick to quick translator cache 131, trap logic circuit 170, software search code 31, 32 and 33 and execution unit 140. Non-quick to quick translator cache 131 includes instruction and data processor 12 and associative memory 14. Associative memory 14, in turn, includes instruction identifier memory section 18, data set memory section 20, input circuit 22 and output circuit 24.

- 17 -

Instruction decoder 135 is coupled to receive a stream of instructions, such as JAVA byte codes, from instruction cache unit 120. Although the present invention is described in connection with JAVA bytecodes, those of skill in the art will recognize variations for dynamic replacement of other types of instructions in other virtual machine environments based on the description herein. Although the bytecode replacement features described herein are generally applicable to instruction execution acceleration based on execution time resolution of instruction parameters as more generally described in the above-incorporated patent application, the description which follows focuses on the dynamic replacement of *pointer non-specific* non-quick bytecodes with *pointer specific* quick variants thereof and on hardware processor 100 for facilities performing this dynamic replacement to facilitate implementation of a write barrier.

Referring to Figure 7, instruction decoder 135 provides decoded bytecodes on bus 11 and program counter (PC) values corresponding to the decoded bytecodes on bus 13. These bytecodes and PC values are provided to execution unit 140 and to instruction and data processor 12. In addition, the PC values are provided to input circuit 22 of associative memory 14. In general, each of the PC values uniquely identifies a corresponding program occurrence of a bytecode. The top entry of operand stack 723 is provided to instruction and data processor 12.

Within associative memory 14, instruction identifier memory section 18 includes multiple (N) entries. Each of these N entries is capable of storing a corresponding bytecode identifier value, such as bytecode identifier values PC\_0, PC\_1, PC\_2, PC\_3, ... PC\_N. Each of the bytecode identifier values stored in instruction identifier memory section 18 corresponds to a different PC value. The width of instruction identifier memory section 18 is selected to correspond with the width of the program counter.

Data set memory section 20 also includes N entries, such that each entry in instruction identifier section 18 has an associated entry in data set section 20. Each of the N entries of data set memory section 20 is capable of storing a data set, such as data sets DATA\_0, DATA\_1, DATA\_2, DATA\_3, ... DATA\_N. As described in more detail below, each of the data sets stored in data set memory section 20 includes data for execution of the quick variant of the corresponding program occurrence of a bytecode. In one embodiment, data set memory section 20 has a width of four 32-bit words. However, data set memory section 20 can have other widths in other embodiments.

Instruction and data processor 12 monitors bytecodes provided on bus 11, and determines whether the current bytecode on bus 11 is a non-quick bytecode which is capable of being executed in an accelerated manner if a corresponding data set is readily accessible. If so, a quick variant of the non-quick bytecode together with its corresponding data set will be cached in non-quick to quick translator cache 131. In general, a non-quick bytecode may have 0, 1, or more quick variants. The JAVA virtual machine specification describes the following non-quick bytecodes: *anewarray*, *checkcast*, *getfield*, *getstatic*, *instanceof*, *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*, *ldc*, *ldc\_w*, *ldc2\_w*, *multianewarray*, *new*, *putfield*, and *putstatic*, which, in one

- 18 -

embodiment of hardware processor 100, have quick variants. For non-quick store-oriented bytecodes, including e.g., `putfield`, `putstatic`, and `aastore`, resolution of constant pool entries corresponding to the target object field allows replacement with a pointer-specific quick variant, e.g., `aputfield_quick` (set reference field in object with garbage collection checks) or

- 5 `aputstatic_quick` (set static reference field in class with garbage collection checks), if the resolution indicates a pointer store operation. If resolution indicates a target object field of a type other than *reference* (i.e., a non-pointer type), replacement is with a different quick variant, e.g., `putfield_quick` (set field in object), `putfield2_quick` (set long or double field in object), `putstatic_quick` (set static field in class), or `putstatic2_quick` (set long or double static field in class).

- 10 In general, non-quick bytecodes which are capable of accelerated execution if a corresponding data set is readily accessible are hereinafter referred to as non-quick bytecodes having quick variants. Non-quick bytecodes having quick variants form a subset of the bytecodes provided by instruction decoder 135. Instruction and data processor 12 determines whether the current bytecode is a non-quick bytecode having a quick variant by decoding an identifying portion (or portions) of the current bytecode. Support is provided by
- 15 entries in instruction identifier memory and data set memory for up to N program occurrences of non-quick bytecodes having quick variants. Some of these entries can be used for pointer-specific quick variants of non-quick store-oriented (but pointer-non-specific) bytecodes for which a corresponding program occurrence resolves to a pointer store.

- Non-quick to quick translator cache 131 operates as follows in response to a current bytecode having
- 20 a current PC value. Instruction decoder 135 provides the current PC value and the decoded current bytecode to execution unit 140 and to instruction and data processor 12. Instruction and data processor 12 is activated when the decoded bytecode is a non-quick bytecode having a quick variant, a quick variant load bytecode, or a retry bytecode. If the current bytecode provided by instruction decoder 135 on bus 11 is not a non-quick bytecode having a quick variant, a quick variant load bytecode or a retry bytecode, then instruction and data
- 25 processor 12 does not respond to the bytecode, and instead, the current bytecode and current PC value are provided to execution unit 140 for execution.

- However, when the current bytecode is a non-quick bytecode having a quick variant, instruction and data processor 12 is activated in response to the current instruction. In one embodiment, bytecodes `putfield` and `putstatic` activate data processor 12. Upon activation, instruction and data processor
- 30 12 determines the status of a signal `NO_MATCH` present on line 21. Initially, the instruction identifier values `PC_0`, `PC_1`, `PC_2`, `PC_3`, ... `PC_N` stored in instruction identifier memory section 18 are set to invalid values. Alternatively, 'valid' bits associated with the instruction identifier values can be cleared. In either case, the current PC value provided to input circuit 22 does not initially match any of the instruction identifier values stored in instruction identifier memory section 18. Consequently, signal `NO_MATCH` is asserted.
- 35 The absence of a match between the current PC value and the instruction identifier values `PC_0`, `PC_1`, `PC_2`,

- 19 -

PC\_3, ... and PC\_N indicates that the data set required to execute the current bytecode is not currently stored in associative memory 14. As a result, instruction and data processor 12 must initially locate and retrieve this data set to allow replacement of the non-quick bytecode with a suitable quick variant.

5 In response to the asserted signal NO\_MATCH and the determination that the current bytecode is a non-quick bytecode having a quick variant, instruction and data processor 12 asserts a control signal TRAP. Control signal TRAP is provided to trap logic 170. In response to control signal TRAP, trap logic 170 temporarily suspends the operation of execution unit 140, and causes a corresponding software code portion 31, 32, or 33 to be accessed. The software code portion accessed is dependent upon the non-quick bytecode which caused control signal TRAP to be asserted.

10 In one embodiment, trap logic 170 accesses instruction cache unit 120 using the current PC value to identify the particular program occurrence of a bytecode which caused control signal TRAP to be asserted. A switch statement implemented in software directs execution to the appropriate software code portion (in response to the identified bytecode). In alternative embodiments, other methods, such as a trap vector, can be used to direct execution to the appropriate software code portion.

15 Thus, when the identified bytecode corresponds to a first bytecode INST\_0, the switch statement causes corresponding software code portion 31 to be accessed. Similarly, when the identified bytecode corresponds to a second bytecode INST\_1, the switch statement causes corresponding software code portion 32 to be accessed. In an exemplary embodiment, first bytecode INST\_0 is putfield and second bytecode INST\_1 is putstatic. When the identified bytecode corresponds to some other bytecode (illustratively designated INST\_N), the switch statement causes a corresponding software code portion 33 to be accessed.

20 Software code portions 31, 32, ... 33 locate and retrieve the data sets required to execute bytecodes INST\_0 (e.g., putfield), INST\_1 (e.g., putstatic), ... INST\_N, respectively. Stated another way, software code portions 31, 32, ... 33 resolve the constant pool entries for program occurrences of bytecodes INST\_0, INST\_1, ... INST\_N, respectively. Because some non-quick bytecodes (e.g., putfield and  
25 putstatic) have multiple quick variants (e.g., putfield\_quick, putfield2\_quick, aputfield\_quick, putstatic\_quick, putstatic2\_quick, and aputstatic\_quick), the corresponding software code portions also select the appropriate quick variant. If resolution of a corresponding constant pool entry indicates a particular program occurrence of a store-oriented bytecode (e.g., putfield) is a pointer store (e.g., if the store target object field is of type  
30 reference), then replacement with a pointer-specific quick variant (e.g., aputfield\_quick) is appropriate.

Software code portions 31, 32, ... 33 further cause the retrieved data sets to be loaded into operand stack 723. Software code portions 31, 32, ... 33 provide quick variant load bytecodes to instruction decoder 135 after the retrieved data sets are loaded into operand stack 723. Instruction decoder 135 decodes the

- 20 -

received quick variant load bytecodes. The decoded quick variant load bytecodes are provided to instruction and data processor 12 on bus 11. Instruction and data processor 12 identifies each quick variant load bytecodes present on bus 11, and in response, retrieves a corresponding data set which was previously loaded into operand stack 723.

5           Instruction and data processor 12 then loads the current PC value and the retrieved data set into associative memory 14. In one example, the current PC value is written to the first entry of instruction identifier memory section 18 as instruction identifier value PC\_0, and the corresponding retrieved data set is written to the first entry of data set section 20 as data set DATA\_0. The current PC value is routed from instruction and data processor 12 to memory section 18 on bus 15. The data set is routed from instruction and data processor 12 to data set memory section 20 on bus 17. The method used to select the particular entry within memory 14 can be, for example, random, a least recently used (LRU) algorithm or a first in, first out (FIFO) algorithm.

          After the current PC value and the retrieved data set have been written to memory 14, instruction and data processor 12 causes the software code to retry the non-quick instruction which caused control signal  
15   **TRAP** to be asserted. At this time, the current PC value, which is again provided to input circuit 22, matches an instruction identifier value (e.g., instruction identifier value PC\_0) stored within the instruction identifier memory section 18. As a result, signal NO\_MATCH is not asserted. Consequently, instruction and data processor 12 does not attempt to locate and retrieve a corresponding data set via trap logic 170 and a corresponding one of software code portions 31, 32 ... 33.

20           Because the current PC value matches instruction identifier value PC\_0, output section 24 passes corresponding data set DATA\_0 to execution unit 140. Consequently, execution unit 140 receives the current PC value and the associated data set DATA\_0 (including the quick variant bytecode) from non-quick to quick translator cache 131. In response, execution unit 140 executes the quick variant bytecode.

          Once the PC value and the data set associated with a non-quick bytecode having a quick variant have  
25   been loaded into associative memory 14, the particular program occurrence of the non-quick bytecode having a quick variant can be subsequently executed without resolving the constant pool and without having to access the software code. Furthermore, for a particular program occurrence of a store-oriented bytecode, a pointer-specific quick variant (e.g., aputfield\_quick) is subsequently executed if the particular program occurrence resolved to a pointer store, and a non-pointer quick variant (e.g., putfield\_quick or  
30   putfield2\_quick) is subsequently executed if the particular program occurrence resolved to a non-pointer (or literal value) store. Moreover, because the non-quick bytecode is not overwritten in the program image, the non-quick bytecode remains available in its original form. In addition, because the non-quick bytecode is not overwritten, the non-quick bytecode can optionally be stored in read only memory.

- 21 -

The following example will further clarify the operation of hardware processor 100, and in particular non-quick to quick translator cache 131 in facilitating a pointer-store-specific embodiment of write barrier 430 for selectively trapping pointer stores by mutator process 410 (Fig. 4). Instruction decoder 135 initially receives non-quick a bytecode (e.g., `putstatic`) having a quick variant, wherein the particular program occurrence of the non-quick bytecode has a corresponding PC value of 0x000100. Assuming that the particular program occurrence of bytecode `putstatic` is not represented in instruction identifier memory section 18, the current PC value of 0x000100 causes input circuit 22 to assert signal `NO_MATCH`. In response to signal `NO_MATCH` and the determination that bytecode `putstatic` is a non-quick bytecode having a quick variant, instruction and data processor 12 asserts control signal `TRAP`. Trap logic 170 uses the PC value to identify the current bytecode as bytecode `INST_1` (i.e., `putstatic`). In response to the current bytecode being identified as bytecode `INST_1`, a software switch statement directs execution to corresponding software code portion 32.

Software code portion 32 then resolves constant pool entries associated with the store target object field, retrieves the data set required to execute bytecode `INST_1`, and loads this data set onto operand stack 723. Software code portion 32 provides a quick variant load bytecode to instruction decoder 135. In response, instruction decoder 135 provides a decoded quick variant load bytecode to instruction and data processor 12. Instruction and data processor 12 retrieves the data set from operand stack 723 and loads this data set into the first entry of data set memory section 20 as data set `DATA_0`. Software code portion 32 determines that the store target object field is of type *reference* (i.e., that the particular program occurrence of `putstatic` is a pointer store) and includes the appropriate pointer-specific quick variant bytecode `aputstatic_quick` with data set `DATA_0`.

Instruction and data processor 12 further loads the current PC value of 0x000100 into the first entry of instruction identifier memory section 18 as instruction identifier value `PC_0`. Instruction and data processor 12 then causes non-quick bytecode `INST_1` (i.e., `putstatic`) and the current PC value of 0x000100 and to be re-asserted on buses 11 and 13, respectively. In one embodiment, instruction and data processor 12 accomplishes this by issuing a return from trap (`ret_from_trap`) bytecode which transfers control back to the bytecode that caused the control signal `TRAP` to be asserted. At this time, input circuit 22 detects a match between the current PC value and instruction identifier value `PC_0`. In response, associative memory 14 provides the data set associated with instruction identifier value `PC_0` (i.e., data set `DATA_0` including the pointer-specific quick variant bytecode `aputstatic_quick`) to output circuit 24. Output circuit 24 passes this data set `DATA_0` to execution unit 140 which executes the pointer-specific quick variant bytecode `aputstatic_quick`.

Other non-quick bytecodes having quick variants and other program instances of the same non-quick bytecode subsequently received by instruction decoder 135 are handled in a similar manner. For example, another program occurrence of the non-quick bytecode `INST_1` (i.e., `putstatic`) having an associated PC

- 22 -

value of 0x000200 can result in the PC value of 0x000200 being stored in instruction identifier section 18 as instruction identifier **PC\_1**, and the data set associated with instruction **INST\_1** being stored in data set memory section 20 as data set **DATA\_1**. If this particular program occurrence of bytecode **putstatic** resolves to a literal value store, the data set associated with instruction identifier value **PC\_1** (i.e., data set **DATA\_1**) will include a quick variant bytecode such as **putstatic2\_quick**, rather than the pointer-specific quick variant. Note that the data set associated with the first program occurrence of non-quick bytecode **INST\_1** (e.g., data set **DATA\_0**) may not be the same as the data set associated with the second program occurrence of non-quick bytecode **INST\_1** (e.g., data set **DATA\_1**).

By resolving the two program occurrences of the bytecode **putstatic**, the first to a pointer-specific store bytecode **aputstatic\_quick** and the second to a non-pointer store bytecode **putstatic2\_quick**, non-quick to quick translator cache 131 of hardware processor 100 restricts write barrier 430 to pointer stores. In one embodiment, the bytecode **aputstatic\_quick** sets a reference field in a target object with garbage collection checks. As described above with reference evaluation of bytecode **aputfield\_quick** by hardware processor 100, write barrier 430 (as implemented by the pointer-specific quick variant bytecode) forms a four-bit index by concatenating the most significant two bits of each of the *objectref* and *value* operand thereto. This index selects a bit from the 16-bit field **WB\_VECTOR** of register **GC\_CONFIG**. If the selected bit is set, a trap **gc\_notify** is generated.

```

    index=((objectref & 0xC0000000)>>28) | (value>>30)
    if (GC_CONFIG.WB_VECTOR[index]==1)
        generate gc_notify

```

In one embodiment of execution unit 140 (Fig. 1), logic circuits for bytecode evaluation couple to register **GC\_CONFIG** (Fig. 4) thereby implementing the above logic expression. Those of skill in the art will recognize a variety of suitable implementations.

In another embodiment, write barrier 430 (as implemented by the pointer-specific quick variant bytecode) supports both intergenerational store trapping and garbage collection page boundary crossing pointer store trapping. As before, this embodiment of write barrier 430 forms a four-bit index by concatenating the most significant two bits of each of the *objectref* and *store\_data* operand thereto. This index selects a bit from the 16-bit field **WB\_VECTOR** of register **GC\_CONFIG**. If the selected bit is set, a trap **gc\_notify** is generated. However, a second trigger is also provided by comparison of masked portions of the *objectref* and *store\_data* operands. The mask is programmably defined by field **GC\_PAGE\_MASK**, i.e., bits 27:16, of register **GC\_CONFIG**. This second trigger is guarded a garbage collection pages enabled bit **GCE** of processor state register **PSR**.



- 23 -

```

if{ (GC_CONFIG[(objectref[31:30]##store_data[31:30])]==1)
  OR ((PSR.GCE==1) AND
      ((store_data[31:12] & 0x3F##GC_CONFIG[27:16]))!=
      (objectref[31:12] & 0x3F##GC_CONFIG[27:16])) }
5      then trap

```

In one embodiment of execution unit 140 (Fig. 1), logic circuits for bytecode evaluation couple to register GC\_CONFIG (Fig. 4) thereby implementing the above logic expression. Those of skill in the art will recognize a variety of suitable implementations.

An advantageous alternative embodiment of write barrier 430 provides a mechanism to restrict  
 10 garbage collection page boundary crossing checks to a particular generation or generations, typically an oldest generation, of a collected memory space. Modified page check trapping equations, e.g.,

```

if(((PSR.GCE==1) ||
    ((objectref[31:30] == store_data[31:30]) &&
    (GEN_PAGE_CHECK_ENABLE[objectref[31:30]]==1))) &&
15 ((objectref[31:12] & 0x3F##GC_PAGE_MASK) !=
    (store_data[31:12] & 0x3F##GC_PAGE_MASK)))
    then trap

```

require that generation tag bits (e.g., bits 31:30) of the *objectref* and *store\_data* operands be equal. To allow flexibility for encoding an oldest generation, four otherwise unused bits of register GC\_CONFIG (e.g., bits  
 20 31:28) can be used to encode field GEN\_PAGE\_CHECK\_ENABLE. This four-bit field indicates which generation or generations to which trapping of garbage collection page boundary crossing stores will be restricted. Those of skill in the art will recognize a variety of suitable implementations, including implementations integrating the generation specific, garbage collection page boundary crossing store trapping with intergenerational pointer store trapping described above.

As described above, operation of non-quick to quick translator cache 131 replaced the original  
 25 bytecode putfield, which was determined to operate on a field having an offset into the class instance data corresponding to a field of type *reference*, with bytecode aputfield\_quick. When the constant pool entry referenced by a putfield instruction was resolved, the offset for the field it referenced was generated and the type of the field was determined to be type *reference*. The 16-bit offset, which is included  
 30 in the corresponding data set DATA\_1 of non-quick to quick translator cache 131, replaces the two operand bytes of the original putfield instruction. The type of the field determined that an aputfield\_quick bytecode, rather than a putfield\_quick or putfield2\_quick bytecode, replaced the original putfield bytecode. Depending on the contents of the fields WB\_VECTOR and

- 24 -

GC\_PAGE\_MASK of register GC\_CONFIG, write barrier 430 (partially implemented in one embodiment by quick variant bytecode `aputfield_quick`) may trap the pointer store as described above.

The forgoing description details one embodiment of dynamic bytecode replacement for hardware processor 100. An alternative embodiment of dynamic bytecode replacement is based on self-modifying code. In light of the description of non-quick to quick translator cache 131, self-modifying code embodiments are advantageous when implementation of a cache is impractical or undesirable (e.g., for cost reasons). In such a case, non-quick to quick translator cache 131 can be eliminated. Instead, trap code, e.g., software code portions 31, 32 ... 33, replaces the original non-quick bytecode with an appropriate quick variant thereof by writing directly into the instruction space so that subsequent executions of the particular program occurrence of the original non-quick bytecode will evaluate the quick variant. One embodiment of a self-modifying code based dynamic bytecode replacement mechanism is described in U.S. Patent No. 5,367,685.

### Garbage Collection Examples

Use of the above-described architectural support for garbage collection is now described in the context of three generational collector approaches: an implementation of Ungar's remembered set generational collector, an implementation of Wilson's card table based generational collector, and an implementation of Hudson's train algorithm-based collector.

#### Remembered Set-based Generational Collector

Figure 8 depicts generational garbage collection using a remembered set after Ungar. An implementation (including e.g., write barrier 430, collector process 420, and garbage collection trap handler 440) of this garbage collection approach can advantageously exploit the features of hardware processor 100 in the following manner:

1. Not all stores need to trap. In particular, only stores of pointers into the heap need to be checked. Use of the above-described `aputfield_quick` and `aputstatic_quick` bytecodes allows checks of only such pointer stores.
- 25 Furthermore, stores to an operand stack or to local variables area represented in stack cache 155 need not trap if the operand stack, local variables, and static areas are assumed to be part of the root set. Object references represented in entries of stack cache 155 can be identified as pointers.
2. As described above, write barrier 430 support of hardware processor 100 traps if the *objectref* of a younger-generation-object is being stored into an older generation object.
- 30 3. In an embodiment in which only two generations are supported within collected memory space 850, field WB\_VECTOR of register GC\_CONFIG contains the value 0x5050. Only one bit of field GC\_TAG (i.e., bit 30) of the *objectref* is considered. In one embodiment, a value of zero (0) identifies the *objectref* as pointing to an object in young generation 810, while a value of one (1)

- 25 -

identifies the *objectref* as pointing to an object in old generation 820. In this embodiment, bit 31 can be effectively ignored. Embodiments for larger numbers of generations will be apparent to those of skill in the art. In accordance with the contents of field WB\_VECTOR, write barrier 430 triggers garbage collection trap handler 440 whenever a pointer from an old generation 820 object into a young generation 810 object is stored. In this embodiment, the PSR.GCE bit is set to zero (0), disabling write barrier 430 operations based on contents of field GC\_PAGE\_MASK.

4. The trap condition for the stores is:

```

if { (GC_CONFIG[ (objectref[31:30]##store_data[31:30]) ] == 1)
    OR ( (PSR.GCE == 1) AND
10      ((store_data[31:12] AND 0x3F##GC_CONFIG[27:16]) !=
        (objectref[31:12] AND 0x3F##GC_CONFIG[27:16]) )
    then trap

```

where *store\_data* is the 32-bit pointer which is being stored into the target object and *objectref* is the 32-bit pointer to the object into which the store is being made.

5. When the hardware processor 100 traps, i.e., when write barrier 430 triggers garbage collection trap handler 440, execution of hardware processor 100 jumps garbage collection trap handler 440. In an embodiment, garbage collection trap handler 440 stores information to remembered set 830 and emulates the trapping pointer store.
6. During garbage collection, objects promoted from young generation 810 to old generation 820, the field GC\_TAG of all references to the promoted object is updated to reflect that the promoted object is part of the older generation.

### Card Table Based Generational Collector

- For a card-based generational collector implementation, the field WB\_VECTOR of register GC\_CONFIG is set to 0xFFFF. This causes write barrier 430 to trap all pointer stores to the heap and trigger a trap handler, such as garbage collection trap handler 440. In such a card-based generational collector embodiment, garbage collection trap handler 440 performs an additional store to a card table data structure and emulates the trapping store. In contrast with a traditional card-based generational collector implementation after Wilson, the embodiment described traps only pointer stores. In an alternative embodiment, field WB\_VECTOR of register GC\_CONFIG is set an appropriate value to define trapping behavior of write barrier 430 corresponding to only pointer stores which are also intergenerational. In this way, collection time scanning can be restricted to cards for which an intergenerational pointer store occurred. In such an embodiment, the card table advantageously provides a duplicate removal function as contrasted with the above-described remembered set embodiment.

### Train Algorithm-Based Collectors

Hudson's train Algorithm is popular for allowing non-disruptive collection of an oldest generation of a generational system. It works using a write barrier to keep track of references between different memory regions ("cars") within the oldest generation. In an hardware processor 100 implementation, these "cars" are defined as fixed, power-of-two sized regions that are aligned on power-of-two boundaries. Field GC\_PAGE\_MASK defines operation of write barrier 430 for this class of garbage collection algorithms. If processor state register bit PSR.GCE is set to one, any pointer stores that cross a garbage collection page boundary (as defined by field GC\_PAGE\_MASK) causes write barrier 430 to trigger garbage collection trap handler 440. In such an embodiment, garbage collection trap handler 440 manages the . Page ("car") size is programmably-defined based on field GC\_PAGE\_MASK of register GC\_CONFIG. Page ranges from 4 KBytes to 8 MBytes are supported in the above-described embodiment.

### Other Collectors

Some real-time garbage collectors that depend on write barriers have been developed. Implementations of these garbage collectors can exploit the pointer specific feature of write barrier 430. Incremental mark-sweep collectors, such as that proposed by Steele (*see* Guy L. Steele, *Multiprocessing Compactifying Garbage Collection*, Communications of the ACM, 18(9) (1975)) can also exploit the pointer specific feature of write barrier 430.

While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Claim terms such as first instruction, second instruction, third instruction, etc. are for identification only and should not be construed to require a particular ordering of instructions. Many variations, modifications, additions, and improvements of the embodiments described are possible. For example, although the present invention has been herein described with reference to exemplary embodiments relating to the JAVA programming language and JAVA virtual machine, it is not limited to them and, instead, encompasses systems, articles, methods, and apparatus for a wide variety of processor environments.

In addition, although certain exemplary embodiments have been described in terms of hardware, software (e.g., interpreter, just-in-time compiler, etc.) implementations of a virtual machine instruction processor employing various of a intergenerational pointer store trap matrix, object reference generation, tagging, a write barrier responsive the intergenerational pointer store trap matrix and object reference generation tagging, a garbage collection trap handler, and/or facilities for selective dynamic replacement of pointer-non-specific instructions with pointer-specific instructions with write barrier support are also suitable. These and other variations, modifications, additions, and improvements may fall within the scope of the invention as defined by the claims which follow.

- 27 -

**WHAT IS CLAIMED IS:**

1. An apparatus comprising:  
a virtual machine instruction processor, wherein instructions executable thereby include program occurrences of a store instruction;  
an instruction replacement component of said virtual machine instruction processor, wherein said instruction replacement component detects said store instruction and selectively replaces a particular program occurrence of said store instruction with a pointer-specific store instruction if a store target field of said particular program occurrence resolves to a pointer-type field; and  
a write barrier provided by execution of said pointer-specific store instruction on said virtual machine instruction processor.
2. An apparatus, as recited in claim 1,  
wherein said instruction replacement component includes a translator cache coupled into an instruction path of said virtual machine instruction processor,  
wherein resolution of said store target field is triggered by said translator cache in response to a program occurrence identifier no match indication;  
wherein said translator cache caches a pointer-specific variant of said store instruction and associates said program occurrence identifier therewith if said resolution indicates that said store target field is of type reference.
3. An apparatus, as recited in claim 2,  
wherein virtual machine instruction processor includes a hardware processor adapted to directly execute at least a subset of said instructions; and  
wherein said translator cache is coupled between an instruction decoder and an execution unit of said hardware processor.
4. An apparatus, as recited in claim 1, wherein said instruction replacement component replaces said particular program occurrence of said store instruction by modifying an in-memory image of said particular program occurrence of said store instruction.
5. An apparatus, as recited in claim 1,  
wherein said virtual machine instruction processor includes a hardware processor adapted to directly execute at least a subset of said instructions; and  
wherein said write barrier includes a logic circuit responsive to a garbage collection configuration register of said hardware processor, said logic circuit employed by said hardware processor to filter garbage collection traps of said pointer-specific store instruction.

- 28 -

1           6.       An apparatus, as recited in claim 1, wherein said virtual machine instruction processor  
2 includes a software program executable on a hardware processor, and wherein said store instruction and said  
3 pointer-specific store instruction are executable by said software program.

1           7.       An apparatus, as recited in claim 6,  
2 wherein said software program defines garbage collection configuration storage accessible thereto;  
3 and  
4 wherein said execution of said pointer-specific store instruction includes evaluation of hardware  
5 processor instructions implementing a logic equation to filter garbage collection traps of said  
6 pointer-specific store instruction in accordance with contents of said garbage collection  
7 configuration storage.

1           8.       An apparatus, as recited in claim 6, wherein said software program includes an interpreter  
2 for said virtual machine instructions.

1           9.       An apparatus, as recited in claim 6, wherein said software program includes a just-in-time  
2 compiler for incrementally compiling said virtual machine instructions to said hardware processor instructions.

1           10.      An apparatus, as recited in claim 1, wherein said store instruction is a JAVA virtual machine  
2 bytecode.

1           11.      An apparatus, as recited in claim 1,  
2 wherein said store instruction is a non-quick bytecode having a first quick variant; and  
3 wherein said pointer-specific instruction is said first quick variant.

1           12.      An apparatus, as recited in claim 5, wherein said garbage configuration register includes an  
2 intergenerational pointer store trap matrix representation having elements corresponding to store target object  
3 and store reference data generation pairs, such that one of said elements indicates whether said write barrier  
4 should trap execution of a particular program occurrence of said pointer-specific store instruction given  
5 generation tags associated with store reference data and a store target object thereof.

1           13.      An apparatus, as recited in claim 1,  
2 wherein said virtual machine instruction processor includes a representation of a garbage collection  
3 configuration store; and  
4 wherein said execution of said pointer specific store instruction includes determining whether a  
5 particular program occurrence of said pointer-specific store instruction will store a boundary

- 29 -

6 crossing pointer indicated as for trapping by said garbage collection configuration store, and,  
7 if so, trapping to a garbage collection fault handler.

1 14. An apparatus, as recited in claim 13,  
2 wherein said boundary crossing pointer is an intergenerational pointer; and  
3 wherein said garbage collection configuration store includes an intergenerational pointer store trap  
4 matrix representation.

1 15. An apparatus, as recited in claim 13,  
2 wherein said boundary crossing pointer is a garbage collection page boundary crossing pointer; and  
3 wherein said garbage collection configuration store includes a garbage collection page mask  
4 representation.

1 16. An apparatus, as recited in claim 1, further comprising:  
2 a fault handler responsive to said write barrier.

1 17. An apparatus, as recited in claim 16, wherein said fault handler includes instructions for  
2 storing to a remembered set.

1 18. An apparatus, as recited in claim 16, wherein said fault handler includes instructions for  
2 storing to a card table.

1 19. An apparatus, as recited in claim 16,  
2 further comprising a generational collector process of instructions executable by said virtual machine  
3 instruction processor; and  
4 wherein said fault handler includes instructions executable by said virtual machine instruction  
5 processor for storing information identifying trapped store data to a data structure for use by  
6 said generational collector process.

1 20. A method for filtering pointer stores, said method comprising:  
2 detecting a program occurrence of a store instruction; and  
3 selectively replacing said program occurrence of said store instruction with a pointer-specific store  
4 instruction based on resolution of store target field type information for said program  
5 occurrence of said store instruction, wherein execution of said pointer-specific store  
6 instruction includes selective trapping in accordance with contents of a garbage collection  
7 configuration store.

- 1        21.        A method, as recited in claim 20, further comprising:  
2        executing said pointer-specific store instruction; and  
3        selectively trapping said executing in accordance with first contents of said garbage collection  
4        configuration store, wherein said garbage collection configuration store programmably  
5        encodes a write barrier to selected intergenerational pointer stores.
- 1        22.        A method, as recited in claim 20, further comprising:  
2        executing said pointer-specific store instruction; and  
3        selectively trapping said executing in accordance with second contents of said garbage collection  
4        configuration store, wherein said garbage collection configuration store programmably  
5        encodes a write barrier to garbage collection page boundary crossing pointer stores.
- 1        23.        A method, as recited in claim 20,  
2        wherein said store instruction is a non-quick bytecode having a first quick variant thereof; and  
3        wherein said pointer-specific store instruction is said first quick variant.
- 1        24.        A method, as recited in claim 20, further comprising:  
2        executing a non pointer store instruction corresponding to a second program occurrence of said store  
3        instruction, wherein execution of said non pointer store instruction does not include selective  
4        trapping in accordance with contents of said garbage collection configuration store.
- 1        25.        A method, as recited in claim 24,  
2        wherein said store instruction is a non-quick bytecode having first and second quick variants thereof;  
3        wherein said pointer-specific store instruction is said first quick variant; and  
4        wherein said non pointer store instruction is said second quick variant.
- 1        26.        A method, as recited in claim 20, further comprising:  
2        responsive to said selective trapping, executing a trap handler.
- 1        27.        A method, as recited in claim 20, further comprising:  
2        responsive to said selective trapping, storing information identifying said store instruction target  
3        object in a remembered set.
- 1        28.        A method, as recited in claim 20, further comprising:  
2        responsive to said selective trapping, storing information indicative of said store instruction target  
3        object in a card table.



- 31 -

1           29.     A method, as recited in claim 20, wherein said detecting includes matching a program  
2 counter value associated with said program occurrence of said store instruction with a stored program counter  
3 value.

1           30.     A method, as recited in claim 20, wherein said selectively replacing includes modifying an  
2 in-memory image of said particular program occurrence of said store instruction.

1           31.     A method, as recited in claim 20, wherein said selectively replacing includes:  
2 performing a lookup in an instruction translator cache using a unique identifier for said program  
3 occurrence of said store instruction; and  
4 if said unique identifier matches a first entry of said instruction translator cache, substituting said  
5 pointer-specific store instruction associated therewith.

1           32.     A method, as recited in claim 31, wherein said selectively replacing further includes:  
2 if said unique identifier matches a second entry of instruction translator cache, substituting a non-  
3 pointer store instruction associated therewith, wherein execution of said non-pointer store  
4 instruction does not include selective trapping in accordance with contents of said garbage  
5 collection configuration store.

1           33.     A method, as recited in claim 31, wherein said selectively replacing further includes:  
2 if said unique identifier does not match any entry of said instruction translator cache, resolving type  
3 information for a store target field of said program occurrence of said store instruction;  
4 storing results of said resolving in said instruction translator cache; and retrying said lookup  
5 performing and said substituting.

1           34.     A method, as recited in claim 20, further comprising:  
2 resolving type information for a store target field of said program occurrence of said store instruction

1           35.     A method, as recited in claim 20, wherein said resolution includes retrieving a data set  
2 associated with said store target field, said data set including said store target field type information.

1           36.     A method, as recited in claim 20, wherein said resolution includes retrieving information,  
2 including said store target field type information, from a constant pool table associated with a store target  
3 object of said program occurrence of said store instruction, said constant pool table including a constant pool  
4 field associated with said store target field.

- 32 -

1           37.     A method, as recited in claim 20, wherein said resolution includes:  
2     retrieving a data set associated with said store target field, said data set including store target field  
3           type information; and  
4     storing said pointer-specific store instruction and information from said data set in an instruction  
5           translator cache.

1           38.     A method for filtering mutator process pointer stores in a virtual machine instruction  
2     processor, said method comprising:  
3           selectively transforming a program occurrence of a pointer non-specific mutator store instruction into  
4           one of a pointer-specific variant and a non-pointer variant thereof, said transforming based  
5           on an execution-time determination of store target field type of said pointer non-specific  
6           mutator store; and  
7     trapping substantially only said pointer-specific variant based on correspondence between operands  
8           thereof and contents of a garbage collection configuration store.

1           39.     A method, as recited in claim 38, further comprising:  
2     subsequent to said transforming and for said transformed program occurrence only, executing said  
3     pointer-specific variant in lieu of said pointer non-specific mutator store instruction.

1           40.     Architectural support for selectively trapping pointer stores in a virtual machine instruction  
2     processor having mutator and garbage collector processes executable thereon, said architectural support  
3     comprising:  
4           a garbage collection configuration store;  
5     instruction replacement means operably coupled into an instruction path of said virtual machine  
6           instruction processor to replace a pointer non-specific instruction with a quick variant  
7           thereof based on resolution of target field type for said pointer non-specific instruction, said  
8           quick variant being a pointer specific quick variant if said target field type is reference type;  
9     a write barrier provided by execution of said pointer specific quick variant on said virtual machine  
10           instruction processor, said write barrier responsive to said garbage collection configuration  
11           store.

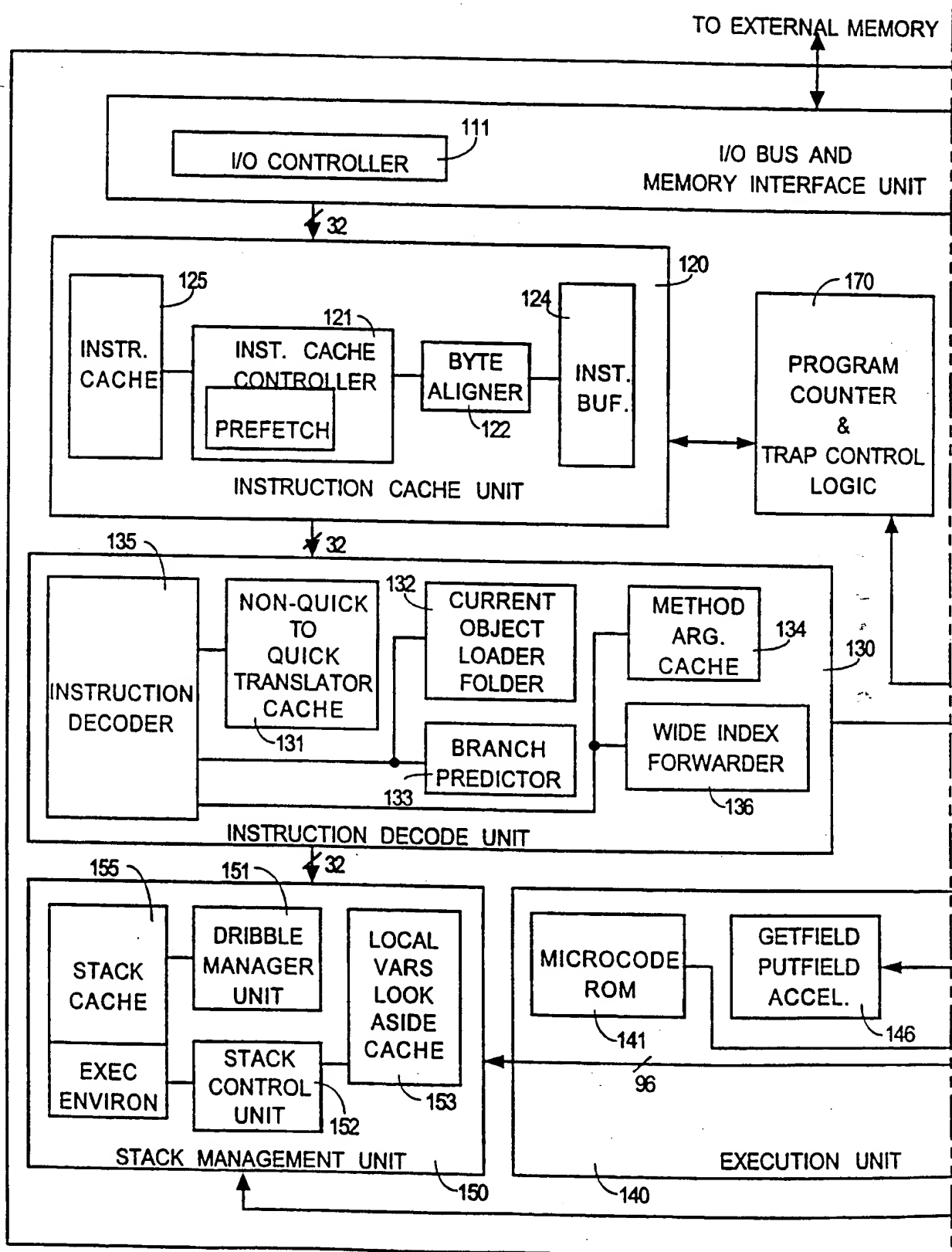
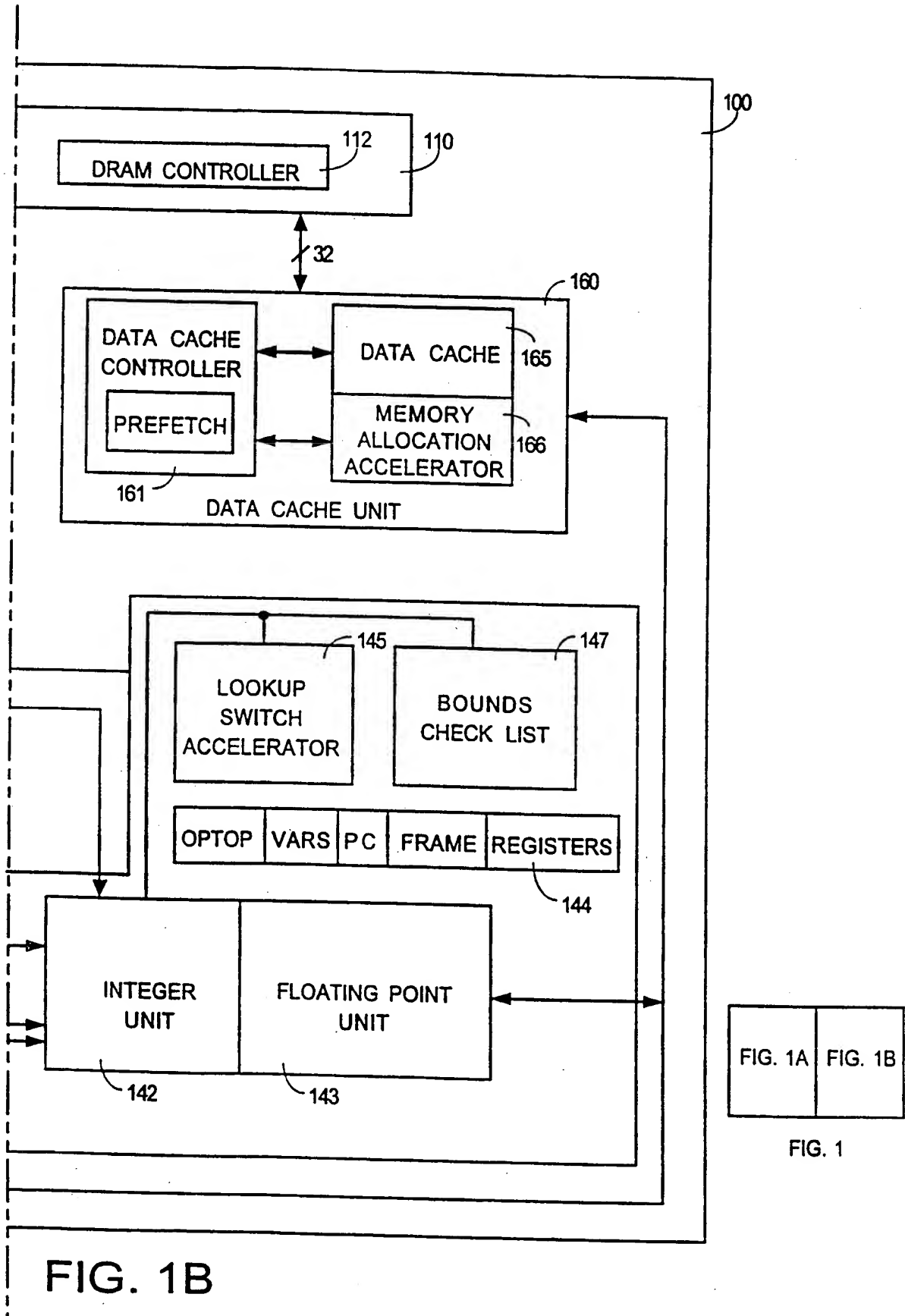


FIG. 1A



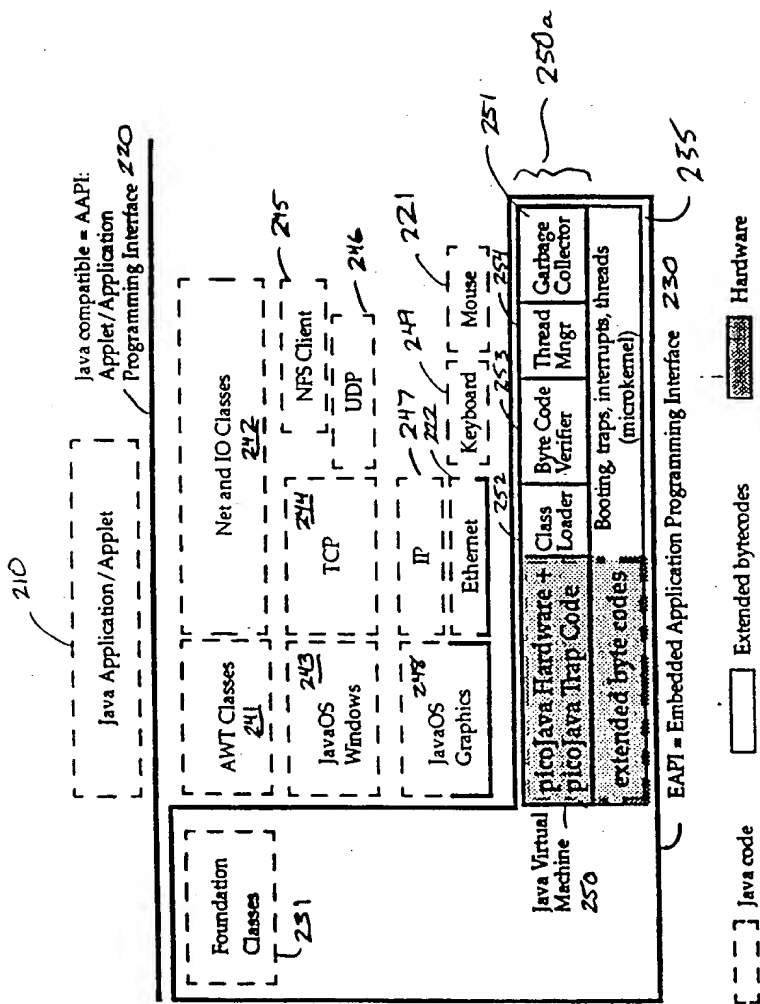
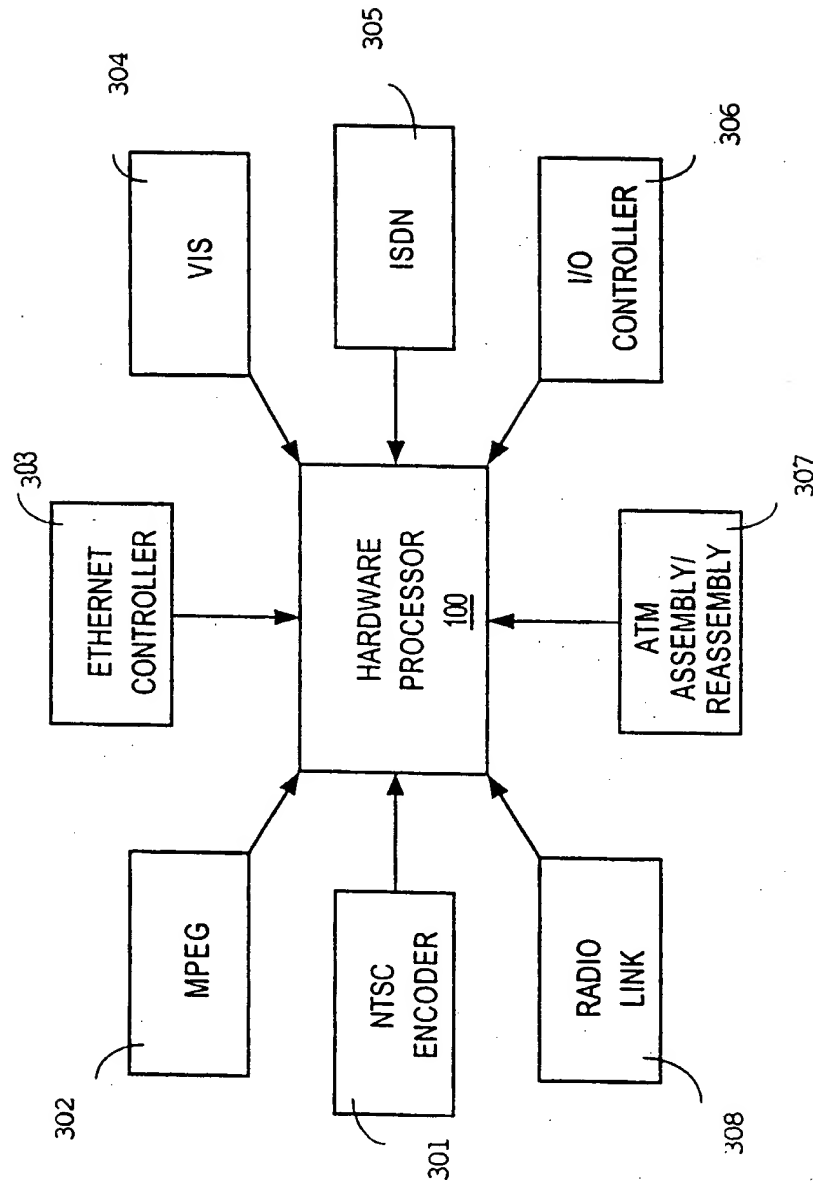


FIG. 2



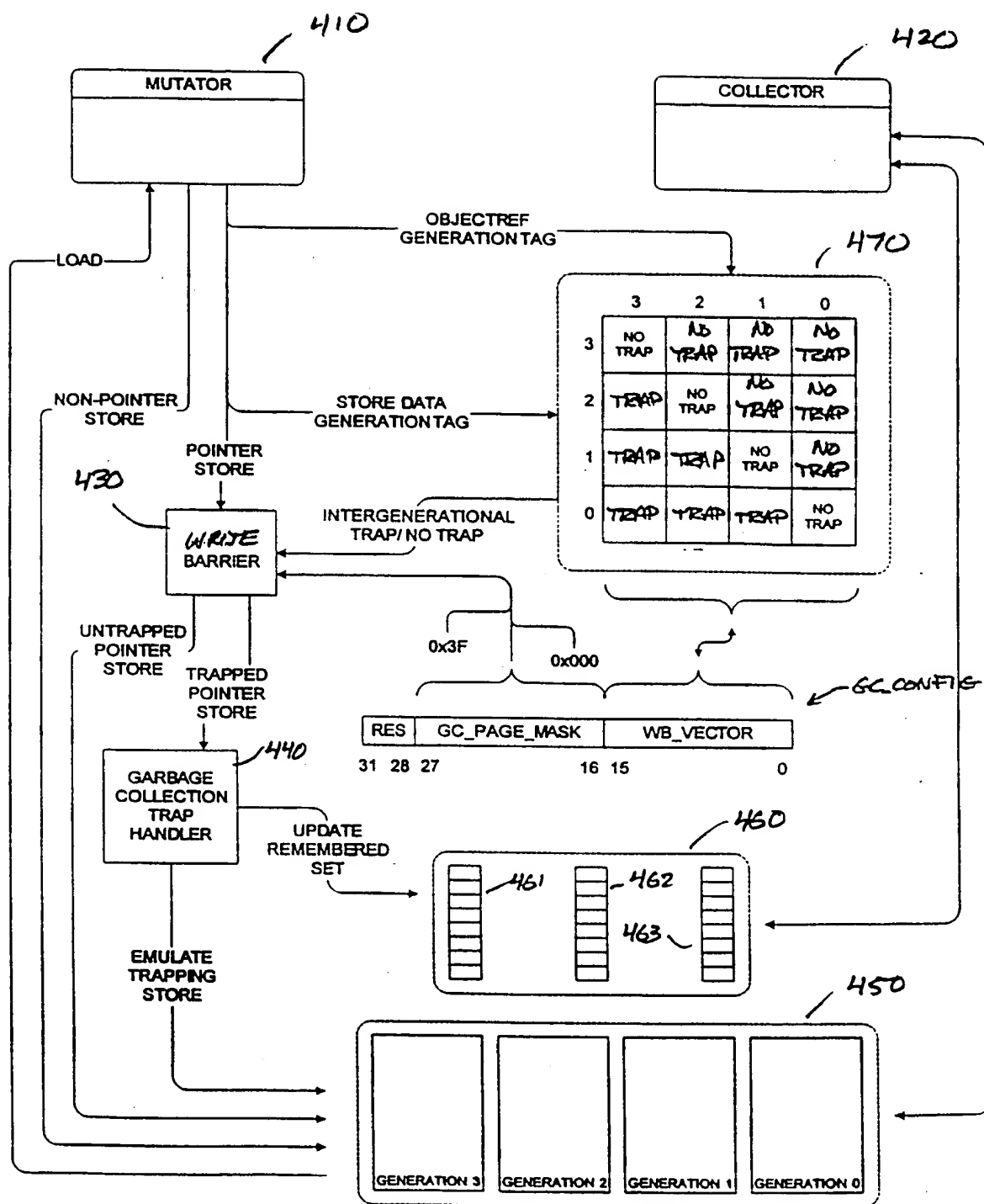


FIGURE 4

500

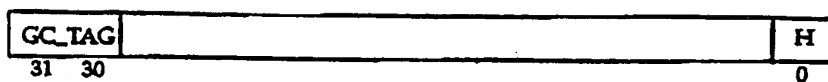


FIG. 5



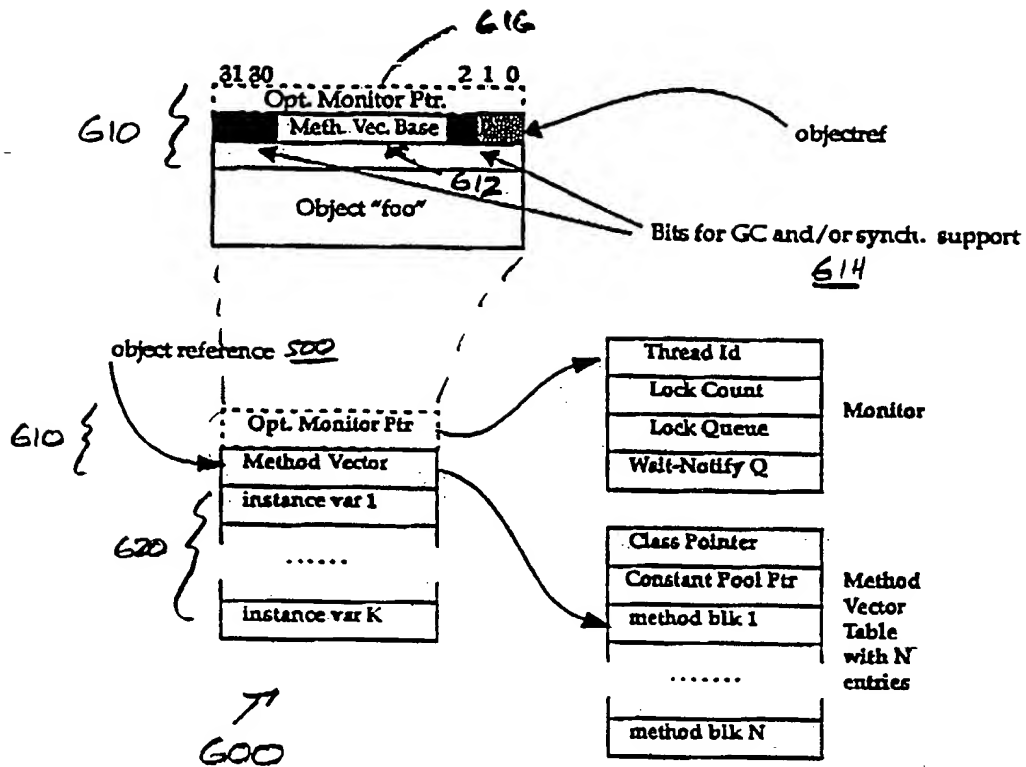


FIG. 6A

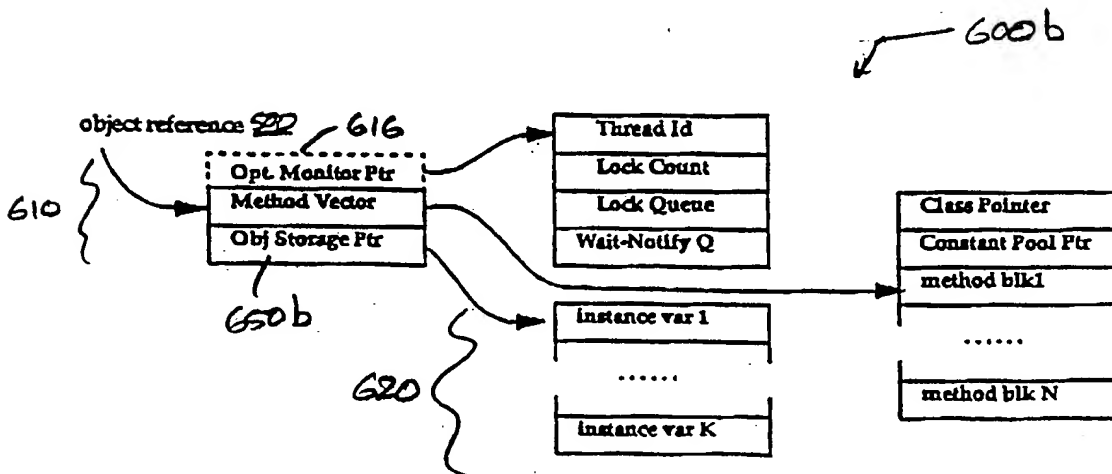


FIG. 6B

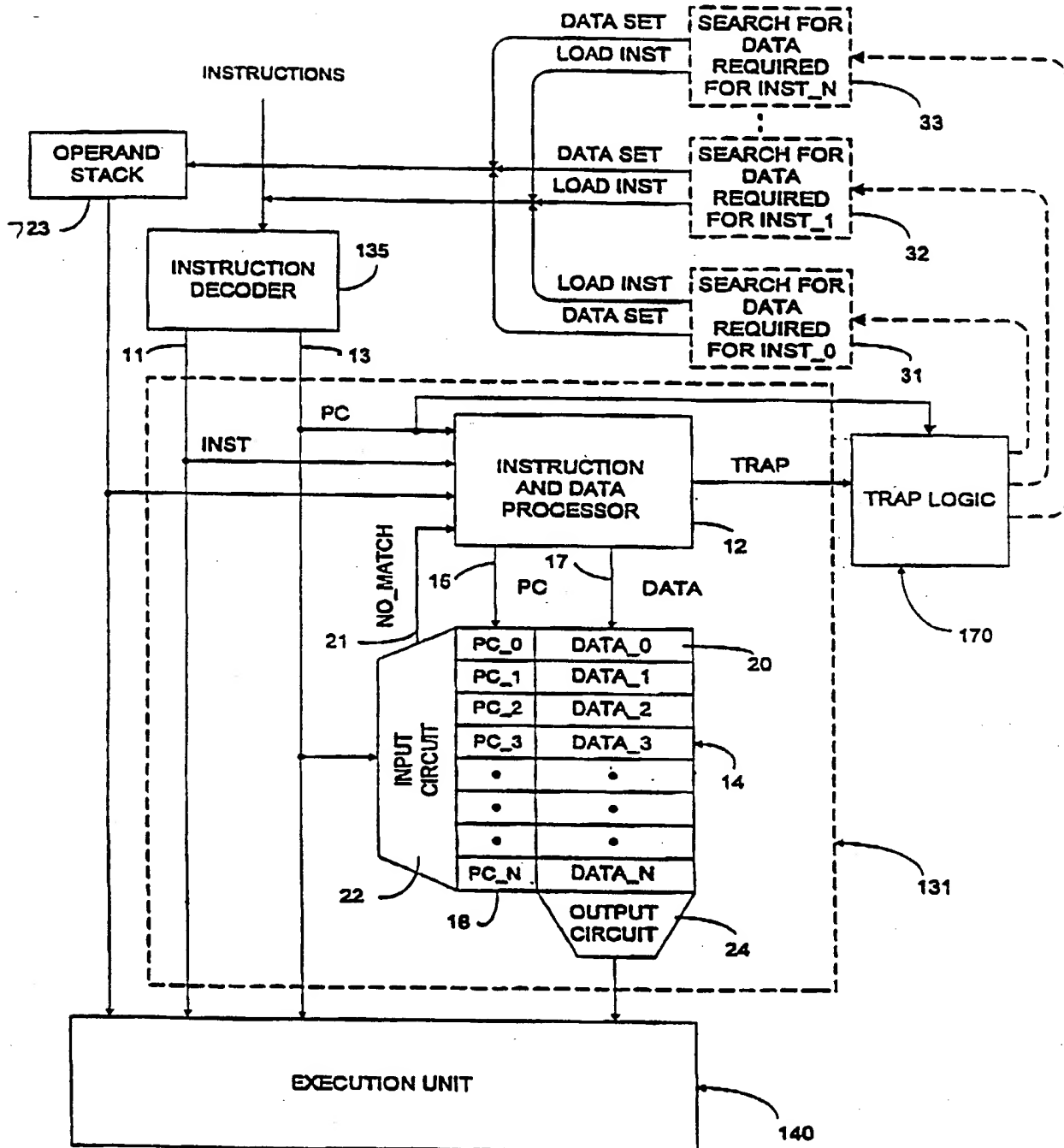


FIGURE 7

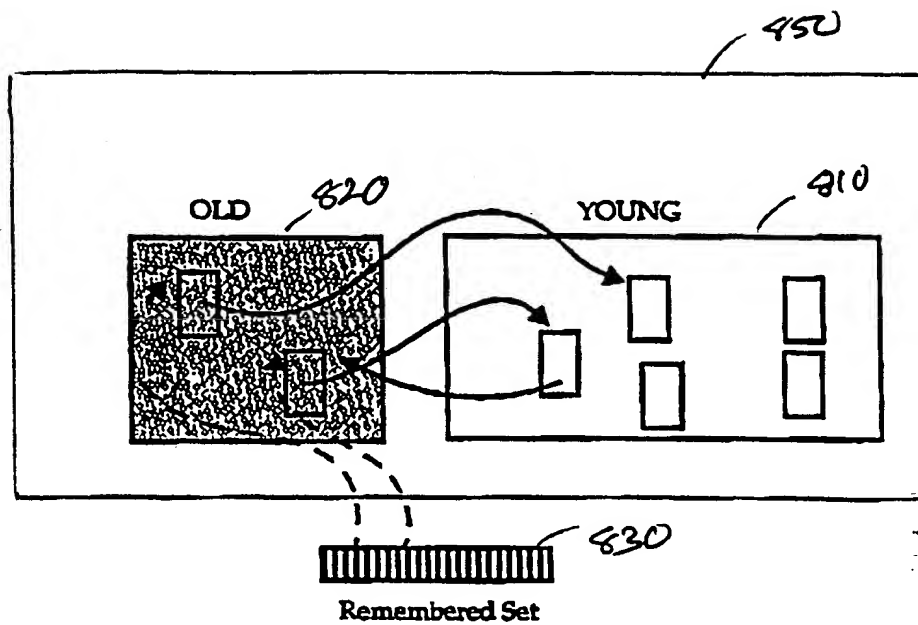


FIG. 8

# INTERNATIONAL SEARCH REPORT

International Application No

PCT/US 98/07624

**A. CLASSIFICATION OF SUBJECT MATTER**  
IPC 6 G06F12/02 G06F9/318

According to International Patent Classification (IPC) or to both national classification and IPC

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)

IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

Category	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 367 685 A (GOSLING JAMES) 22 November 1994 cited in the application see column 2, line 22 - line 68 ---	1, 4, 20, 38, 40
A	HOSKING A L ET AL: "A COMPARATIVE PERFORMANCE EVALUATION OF WRITE BARRIER IMPLEMENTATIONS" ACM SIGPLAN NOTICES, vol. 27, no. 10, 1 October 1992, pages 92-109, XP000327291 see page 96, left-hand column, paragraph 3.1.1 --- -/-	1, 5-7, 12-22, 38, 40



Further documents are listed in the continuation of box C.



Patent family members are listed in annex.

**Special categories of cited documents**

- "A" document defining the general state of the art which is not considered to be of particular relevance
- "E" earlier document but published on or after the international filing date
- "L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
- "O" document referring to an oral disclosure, use, exhibition or other means
- "P" document published prior to the international filing date but later than the priority date claimed

- "T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
- "X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
- "Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
- "&" document member of the same patent family

Date of the actual completion of the international search

24 July 1998

Date of mailing of the international search report

03/08/1998

Name and mailing address of the ISA

European Patent Office, P.B. 5818 Patentlaan 2  
NL - 2280 HV Rijswijk  
Tel. (+31-70) 340-2040. Tx. 31 651 epo nl.  
Fax: (+31-70) 340-3016

Authorized officer

Nielsen, O

# INTERNATIONAL SEARCH REPORT

In ternational Application No

PCT/US 98/07624

## C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	<p>EP 0 737 914 A (IBM) 16 October 1996</p> <p>see abstract  see page 9, line 25 - page 10, line 18;  figures 5,6</p> <p>-----</p>	<p>1-3, 20,  31-33,  38-40</p>

# INTERNATIONAL SEARCH REPORT

Information on patent family members

International Application No

PCT/US 98/07624

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5367685 A	22-11-1994	EP 0604002 A	29-06-1994
		JP 6230976 A	19-08-1994
EP 0737914 A	16-10-1996	US 5619665 A	08-04-1997
		JP 8297572 A	12-11-1996